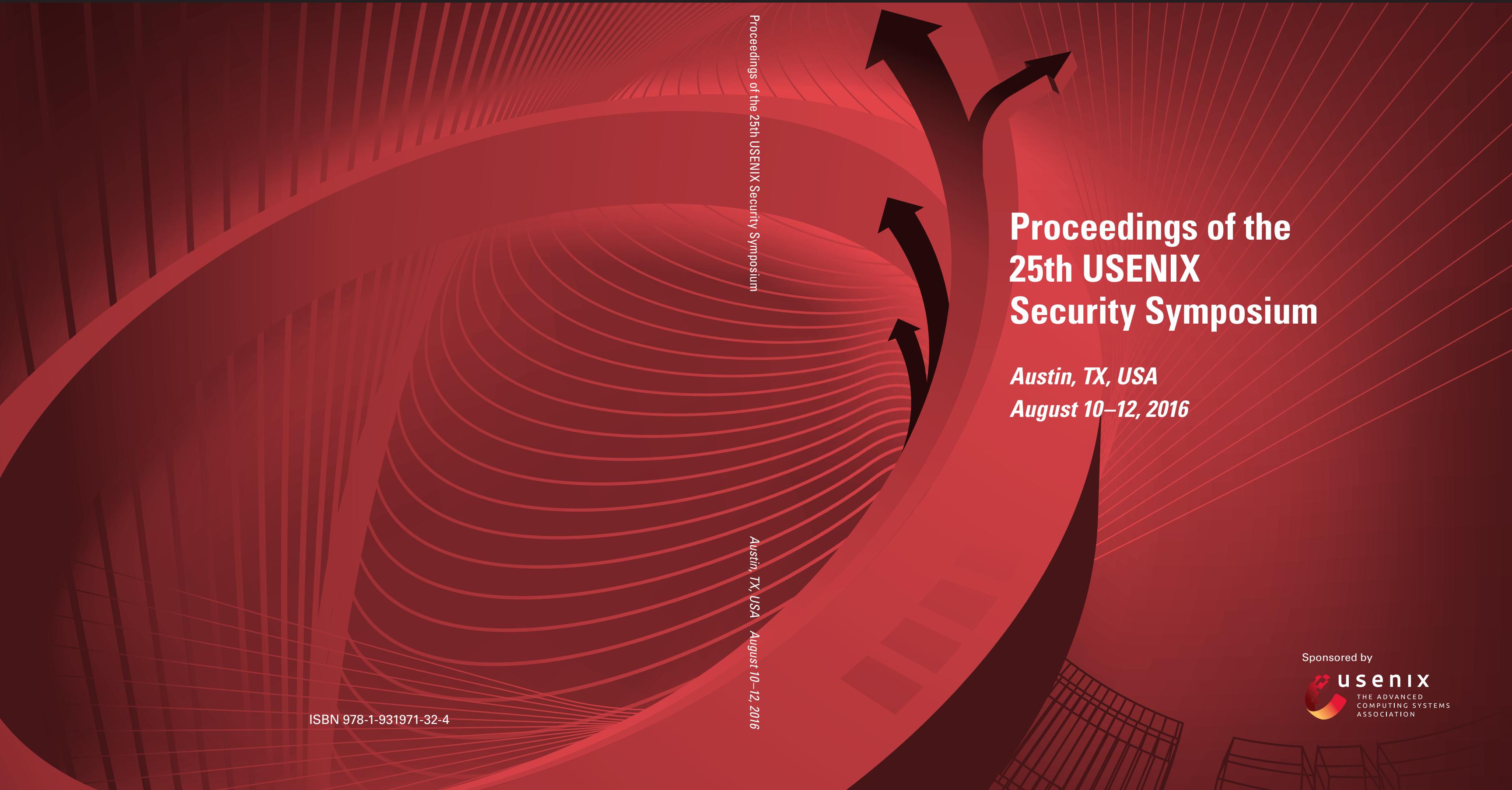


Proceedings of the 25th USENIX Security Symposium

Austin, TX, USA

August 10–12, 2016



ISBN 978-1-931971-32-4

Proceedings of the 25th USENIX Security Symposium

Austin, TX, USA August 10–12, 2016

Sponsored by

THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Thanks to Our USENIX Security '16 Sponsors

Platinum Sponsor



Gold Sponsor



Silver Sponsors



Bronze Sponsors



Media Sponsors and Industry Partners

ACM Queue

Linux Pro Magazine

ADMIN

LXer

Blacks In Technology

No Starch Press

CRC Press

O'Reilly Media

Distributed Management

UserFriendly.Org

Task Force (DMTF)

Virus Bulletin

© 2016 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-931971-32-4

Thanks to Our USENIX Supporters

USENIX Patrons

Facebook Google Microsoft NetApp VMware

USENIX Benefactors

ADMIN *Linux Pro Magazine*

USENIX Partners

Booking.com Can Stock Photo

Open Access Publishing Partner

PeerJ

USENIX Association

**Proceedings of the
25th USENIX Security Symposium**

**August 10–12, 2016
Austin, TX**

Conference Organizers

Program Co-Chairs

Thorsten Holz, *Ruhr-Universität Bochum*
Stefan Savage, *University of California, San Diego*

Program Committee

Michael Backes, *CISPA, Saarland University, and MPI-SWS*
Michael Bailey, *University of Illinois at Urbana–Champaign*
Davide Balzarotti, *Eurecom*
Lujo Bauer, *Carnegie Mellon University*
Leyla Bilge, *Symantec*
Dan Boneh, *Stanford University*
Joseph Bonneau, *Stanford University and The Electronic Frontier Foundation*
Nikita Borisov, *University of Illinois at Urbana–Champaign*
Elie Bursztein, *Google*
Juan Caballero, *IMDEA Software Institute*
Srdjan Capkun, *ETH Zurich*
Stephen Checkoway, *University of Illinois at Chicago*
Nicolas Christin, *Carnegie Mellon University*
Manuel Costa, *Microsoft Research*
George Danezis, *University College London*
Tamara Denning, *University of Utah*
Adam Doupé, *Arizona State University*
Tudor Dumitras, *University of Maryland, College Park*
Manuel Egele, *Boston University*
Serge Egelman, *University of California, Berkeley, and International Computer Science Institute*
David Evans, *University of Virginia*
Cédric Fournet, *Microsoft Research*
Matthew Fredrikson, *Carnegie Mellon University*
Cristiano Giuffrida, *Vrije Universiteit Amsterdam*
Matthew Green, *Johns Hopkins University*
Chris Grier, *Databricks*
Guofei Gu, *Texas A&M University*
Saikat Guha, *Microsoft Research India*
Alex Halderman, *University of Michigan*
Nadia Heninger, *University of Pennsylvania*
Cynthia Irvine, *Naval Postgraduate School*
Martin Johns, *SAP Research*
Engin Kirda, *Northeastern University*
Tadayoshi Kohno, *University of Washington*
Farinaz Koushanfar, *University of California, San Diego*
Per Larsen, *University of California, Irvine*
Wenke Lee, *Georgia Tech*
Nektarios Leontiadis, *Facebook*
Janne Lindqvist, *Rutgers University*
Ben Livshits, *Microsoft Research*
Michelle Mazurek, *University of Maryland, College Park*
Stephen McCamant, *University of Minnesota*
Damon McCoy, *New York University/ICSI*
Jonathan McCune, *Google*
Sarah Meiklejohn, *University College London*
Prateek Mittal, *Princeton University*
Tyler Moore, *University of Tulsa*
Arvind Narayanan, *Princeton University*
Nick Nikiforakis, *Stony Brook University*
Cristina Nita-Rotaru, *Northeastern University*
Mathias Payer, *Purdue University*
Zachary N. J. Peterson, *California Polytechnic State University*
Frank Piessens, *Katholieke Universiteit Leuven*
Michalis Polychronakis, *Stony Brook University*
Raluca Popa, *University of California, Berkeley*
Christina Pöpper, *New York University*
Adrienne Porter Felt, *Google*
Georgios Portokalidis, *Stevens Institute of Technology*
Niels Provos, *Google*
Tom Ristenpart, *Cornell Tech*
Will Robertson, *Northeastern University*
Franziska Roesner, *University of Washington*
Andrei Sabelfeld, *Chalmers University of Technology*
Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*
Felix Schuster, *Microsoft Research*
Jörg Schwenk, *Ruhr-Universität Bochum*
Hovav Shacham, *University of California, San Diego*
Micah Sherr, *Georgetown University*
Elaine Shi, *University of Maryland, College Park*
Reza Shokri, *The University of Texas at Austin*
Deian Stefan, *University of California, San Diego*
Gianluca Stringhini, *University College London*
Cynthia Sturton, *The University of North Carolina at Chapel Hill*
Kurt Thomas, *Google*
Patrick Traynor, *University of Florida*
Giovanni Vigna, *University of California, Santa Barbara*
David Wagner, *University of California, Berkeley*
Nick Weaver, *International Computer Science Institute*

Invited Talks Chair

Adrienne Porter Felt, *Google*

Invited Talks Committee

Tyrone Grandison, *US Department of Commerce*
Alex Halderman, *University of Michigan*
Franziska Roesner, *University of Washington*
Elaine Shi, *Cornell University*

Poster Session Chair

Raluca Popa, *University of California, Berkeley*

Poster Session Committee Members

Nikita Borisov, *University of Illinois at Urbana-Champaign*
Mathias Payer, *Purdue University*

Steering Committee

Matt Blaze, *University of Pennsylvania*
Dan Boneh, *Stanford University*
Kevin Fu, *University of Michigan*
Casey Henderson, *USENIX Association*
Jaeyeon Jung, *Microsoft Research*
Tadayoshi Kohno, *University of Washington*
Niels Provos, *Google*
David Wagner, *University of California, Berkeley*
Dan Wallach, *Rice University*

External Reviewers

David Adrian

Marc Andryesco

Musard Balliu

Nataliia Bielova

Shaanan Cohney

Antoine Delignat-Lavaud

Zakir Durumeric

Per Hallgren

Marcella Hastings

Daniel Hausknecht

Daniel Hedin

Brett Hemenway

Kevin Hong

Sanghyun Hong

Soumya Indela

Kai Jansen

David Kohlbrenner

Katharina Kohls

Srijan Kumar

BumJun Kwon

Philip Mackenzie

John Manferdelli

Abner Mendoza

Jan-Tobias Muehlberg

Job Noorman

Martin Ochoa

Moheeb Abu Rajab

David Rupprecht

Daniel Schoepe

Alexander Sjösten

Drew Springall

Rock Stevens

Raoul Strackx

Octavian Suciu

Hamid Ebadi Tavallaei

Luke Valenta

Steven Van Acker

Jo Van Bulck

Neline van Ginkel

Haopei Wang

Eric Wustrow

Lei Xu

Guangliang Yang

Jialong Zhang

Ziyun Zhu

25th USENIX Security Symposium
August 10–12, 2016
Austin, TX

Message from the Program Co-Chairs.....x

Wednesday, August 10

Low-Level Attacks

Flip Feng Shui: Hammering a Needle in the Software Stack	1
Kaveh Razavi, Ben Gras, and Erik Bosman, <i>Vrije Universiteit Amsterdam</i> ; Bart Preneel, <i>Katholieke Universiteit Leuven</i> ; Cristiano Giuffrida and Herbert Bos, <i>Vrije Universiteit Amsterdam</i>	
One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation	19
Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu, <i>The Ohio State University</i>	
PIkit: A New Kernel-Independent Processor-Interconnect Rootkit	37
Wonjun Song, Hyunwoo Choi, Junhong Kim, Eunsoo Kim, Yongdae Kim, and John Kim, <i>Korea Advanced Institute of Science and Technology (KAIST)</i>	

Verification and Timing

Verifying Constant-Time Implementations	53
José Bacelar Almeida, <i>HASLab/INESC TEC and University of Minho</i> ; Manuel Barbosa, <i>HASLab/INESC TEC and DCC FCUP</i> ; Gilles Barthe and François Dupressoir, <i>IMDEA Software Institute</i> ; Michael Emmi, <i>Bell Labs and Nokia</i>	
Secure, Precise, and Fast Floating-Point Operations on x86 Processors	71
Ashay Rane, Calvin Lin, and Mohit Tiwari, <i>The University of Texas at Austin</i>	
ÜBERSPARK: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor	87
Amit Vasudevan and Sagar Chaki, <i>Carnegie Mellon University</i> ; Petros Maniatis, <i>Google Inc.</i> ; Limin Jia and Anupam Datta, <i>Carnegie Mellon University</i>	

Software Attacks

Undermining Information Hiding (and What to Do about It)	105
Enes Göktaş, <i>Vrije Universiteit Amsterdam</i> ; Robert Gawlik and Benjamin Kollenda, <i>Ruhr Universität Bochum</i> ; Elias Athanasopoulos, <i>Vrije Universiteit Amsterdam</i> ; Georgios Portokalidis, <i>Stevens Institute of Technology</i> ; Cristiano Giuffrida and Herbert Bos, <i>Vrije Universiteit Amsterdam</i>	
Poking Holes in Information Hiding.....	121
Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida, <i>Vrije Universiteit Amsterdam</i>	
What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses.....	139
Giorgi Maisuradze, Michael Backes, and Christian Rossow, <i>Saarland University</i>	

Password and Key-Fingerprints

zxcvbn: Low-Budget Password Strength Estimation	157
Daniel Lowe Wheeler, <i>Dropbox Inc.</i>	
Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks	175
William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor, <i>Carnegie Mellon University</i>	

An Empirical Study of Textual Key-Fingerprint Representations	193
Sergej Dechand, <i>University of Bonn</i> ; Dominik Schürmann, <i>Technische Universität Braunschweig</i> ; Karoline Busse, <i>University of Bonn</i> ; Yasemin Acar and Sascha Fahl, <i>Saarland University</i> ; Matthew Smith, <i>University of Bonn</i>	

Network Security

Off-Path TCP Exploits: Global Rate Limit Considered Dangerous	209
Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, and Srikanth V. Krishnamurthy, <i>University of California, Riverside</i> ; Lisa M. Marvel, <i>United States Army Research Laboratory</i>	

Website-Targeted False Content Injection by Network Operators	227
Gabi Nakibly, <i>Rafael—Advanced Defense Systems and Technion—Israel Institute of Technology</i> ; Jaime Schcolnik, <i>Interdisciplinary Center Herzliya</i> ; Yossi Rubin, <i>Rafael—Advanced Defense Systems</i>	

The Ever-Changing Labyrinth: A Large-Scale Analysis of Wildcard DNS Powered Blackhat SEO	245
Kun Du and Hao Yang, <i>Tsinghua University</i> ; Zhou Li, <i>IEEE Member</i> ; Haixin Duan, <i>Tsinghua University</i> ; Kehuan Zhang, <i>The Chinese University of Hong Kong</i>	

A Comprehensive Measurement Study of Domain Generating Malware.....	263
Daniel Plohmann, <i>Fraunhofer FKIE</i> ; Khaled Yakdan, <i>University of Bonn</i> ; Michael Klatt, <i>DomainTools</i> ; Johannes Bader; Elmar Gerhards-Padilla, <i>Fraunhofer FKIE</i>	

Applied Cryptography

Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing	279
Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford, <i>École Polytechnique Fédérale de Lausanne (EPFL)</i>	

Faster Malicious 2-Party Secure Computation with Online/Offline Dual Execution	297
Peter Rindal and Mike Rosulek, <i>Oregon State University</i>	

Egalitarian Computing.....	315
Alex Biryukov and Dmitry Khovratovich, <i>University of Luxembourg</i>	

Post-quantum Key Exchange—A New Hope	327
Erdem Alkim, <i>Ege University</i> ; Léo Ducas, <i>Centrum voor Wiskunde en Informatica</i> ; Thomas Pöppelmann, <i>Infineon Technologies AG</i> ; Peter Schwabe, <i>Radboud University</i>	

Thursday, August 11

Software Security

Automatically Detecting Error Handling Bugs Using Error Specifications.....	345
Suman Jana and Yuan Kang, <i>Columbia University</i> ; Samuel Roth, <i>Ohio Northern University</i> ; Baishakhi Ray, <i>University of Virginia</i>	

APISAN: Sanitizing API Usages through Semantic Cross-Checking	363
Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik, <i>Georgia Institute of Technology</i>	

On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re)introduces Software Vulnerabilities.....	379
Santiago Torres-Arias, <i>New York University</i> ; Anil Kumar Ammula and Reza Curtmola, <i>New Jersey Institute of Technology</i> ; Justin Cappos, <i>New York University</i>	

(Thursday, August 11, continues on next page)

Hardware I

Defending against Malicious Peripherals with Cinch397
Sebastian Angel, <i>The University of Texas at Austin and New York University</i> ; Riad S. Wahby, <i>Stanford University</i> ; Max Howald, <i>The Cooper Union and New York University</i> ; Joshua B. Leners, <i>Two Sigma</i> ; Michael Spilo and Zhen Sun, <i>New York University</i> ; Andrew J. Blumberg, <i>The University of Texas at Austin</i> ; Michael Walfish, <i>New York University</i>	

Making USB Great Again with USBFILTER415
--	-------------

Dave (Jing) Tian and Nolen Scaife, *University of Florida*; Adam Bates, *University of Illinois at Urbana–Champaign*; Kevin R. B. Butler and Patrick Traynor, *University of Florida*

Micro-Virtualization Memory Tracing to Detect and Prevent Spraying Attacks431
---	-------------

Stefano Cristalli and Mattia Pagnozzi, *University of Milan*; Mariano Graziano, *Cisco Systems Inc.*; Andrea Lanzi, *University of Milan*; Davide Balzarotti, *Eurecom*

Web Security

Request and Conquer: Exposing Cross-Origin Resource Size447
---	-------------

Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen, *Katholieke Universiteit Leuven*

Trusted Browsers for Uncertain Times463
---	-------------

David Kohlbrenner and Hovav Shacham, *University of California, San Diego*

Tracing Information Flows Between Ad Exchanges Using Retargeted Ads481
--	-------------

Muhammad Ahmad Bashir, Sajjad Arshad, William Robertson, and Christo Wilson, *Northeastern University*

Cyber-Physical Systems

Virtual U: Defeating Face Liveness Detection by Building Virtual Models from Your Public Photos497
--	-------------

Yi Xu, True Price, Jan-Michael Frahm, and Fabian Monrose, *The University of North Carolina at Chapel Hill*

Hidden Voice Commands513
------------------------------------	-------------

Nicholas Carlini and Pratyush Mishra, *University of California, Berkeley*; Tavish Vaidya, Yuankai Zhang, Micah Sherr, and Clay Shields, *Georgetown University*; David Wagner, *University of California, Berkeley*; Wenchao Zhou, *Georgetown University*

FlowFence: Practical Data Protection for Emerging IoT Application Frameworks531
---	-------------

Earlene Fernandes, Justin Paupore, and Amir Rahmati, *University of Michigan*; Daniel Simionato and Mauro Conti, *University of Padova*; Atul Prakash, *University of Michigan*

Low-Level Attacks and Defenses

ARMageddon: Cache Attacks on Mobile Devices549
--	-------------

Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard, *Graz University of Technology*

DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks565
--	-------------

Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard, *Graz University of Technology*

An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries583
---	-------------

Dennis Andriesse, Xi Chen, and Victor van der Veen, *Vrije Universiteit Amsterdam*; Asia Slowinska, *Lastline, Inc.*; Herbert Bos, *Vrije Universiteit Amsterdam*

Machine Learning and Data Retrieval Systems

Stealing Machine Learning Models via Prediction APIs601
---	-------------

Florian Tramèr, *École Polytechnique Fédérale de Lausanne (EPFL)*; Fan Zhang, *Cornell University*; Ari Juels, *Cornell Tech*; Michael K. Reiter, *The University of North Carolina at Chapel Hill*; Thomas Ristenpart, *Cornell Tech*

Oblivious Multi-Party Machine Learning on Trusted Processors	619
Olga Ohrimenko, Felix Schuster, and Cédric Fournet, <i>Microsoft Research</i> ; Aastha Mehta, <i>Microsoft Research and Max Planck Institute for Software Systems (MPI-SWS)</i> ; Sebastian Nowozin, Kapil Vaswani, and Manuel Costa, <i>Microsoft Research</i>	

Thoth: Comprehensive Policy Compliance in Data Retrieval Systems	637
Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel, <i>Max Planck Institute for Software Systems (MPI-SWS)</i>	

Crypto Attacks

Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage.....	655
Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan, <i>Johns Hopkins University</i>	

Predicting, Decrypting, and Abusing WPA2/802.11 Group Keys	673
Mathy Vanhoef and Frank Piessens, <i>Katholieke Universiteit Leuven</i>	

DROWN: Breaking TLS using SSLv2	689
Nimrod Aviram, <i>Tel Aviv University</i> ; Sebastian Schinzel, <i>Münster University of Applied Sciences</i> ; Juraj Somorovsky, <i>Ruhr University Bochum</i> ; Nadia Heninger, <i>University of Pennsylvania</i> ; Maik Dankel, <i>Münster University of Applied Sciences</i> ; Jens Steube, <i>Hashcat Project</i> ; Luke Valenta, <i>University of Pennsylvania</i> ; David Adrian and J. Alex Halderman, <i>University of Michigan</i> ; Viktor Dukhovni, <i>Two Sigma and OpenSSL</i> ; Emilia Käuper, <i>Google and OpenSSL</i> ; Shaanan Cohney, <i>University of Pennsylvania</i> ; Susanne Engels and Christof Paar, <i>Ruhr University Bochum</i> ; Yuval Shavitt, <i>Tel Aviv University</i>	

All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption.....	707
Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou, <i>University of Maryland</i>	

Malware

Investigating Commercial Pay-Per-Install and the Distribution of Unwanted Software	721
Kurt Thomas, Juan A. Elices Crespo, Ryan Rasti, Jean-Michel Picod, Cait Phillips, Marc-André Decoste, Chris Sharp, Fabio Tirelo, Ali Tofigh, Marc-Antoine Courteau, Lucas Ballard, Robert Shield, Nav Jagpal, Moheeb Abu Rajab, Panayiotis Mavrommatis, Niels Provos, and Elie Bursztein, <i>Google</i> ; Damon McCoy, <i>New York University and International Computer Science Institute</i>	

Measuring PUP Prevalence and PUP Distribution through Pay-Per-Install Services	739
Platon Kotzias, <i>IMDEA Software Institute and Universidad Politécnica de Madrid</i> ; Leyla Bilge, <i>Symantec Research Labs</i> ; Juan Caballero, <i>IMDEA Software Institute</i>	

UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware	757
Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda, <i>Northeastern University</i>	

Towards Measuring and Mitigating Social Engineering Software Download Attacks	773
Terry Nelms, <i>Georgia Institute of Technology and Damballa</i> ; Roberto Perdisci, <i>University of Georgia and Georgia Institute of Technology</i> ; Manos Antonakakis, <i>Georgia Institute of Technology</i> ; Mustaque Ahamed, <i>Georgia Institute of Technology and New York University Abu Dhabi</i>	

Friday, August 12

Network Security II

Specification Mining for Intrusion Detection in Networked Control Systems	791
Marco Caselli, <i>University of Twente</i> ; Emmanuele Zambon, <i>University of Twente and SecurityMatters B.V.</i> ;	
Johanna Amann, <i>International Computer Science Institute</i> ; Robin Sommer, <i>International Computer Science Institute and Lawrence Berkeley National Laboratory</i> ; Frank Kargl, <i>Ulm University</i>	
Optimized Invariant Representation of Network Traffic for Detecting Unseen Malware Variants	807
Karel Bartos and Michal Sofka, <i>Cisco Systems, Inc.</i> ; Vojtech Franc, <i>Czech Technical University in Prague</i>	
Authenticated Network Time Synchronization	823
Benjamin Dowling, <i>Queensland University of Technology</i> ; Douglas Stebila, <i>McMaster University</i> ; Greg Zaverucha, <i>Microsoft Research</i>	

Hardware II

fTPM: A Software-Only Implementation of a TPM Chip	841
Himanshu Raj, <i>ContainerX</i> ; Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten, <i>Microsoft</i>	
Sanctum: Minimal Hardware Extensions for Strong Software Isolation	857
Victor Costan, Ilia Lebedev, and Srinivas Devadas, <i>MIT CSAIL</i>	
Ariadne: A Minimal Approach to State Continuity	875
Raoul Strackx and Frank Piessens, <i>Katholieke Universiteit Leuven</i>	

Cyber-Physical Systems II

The Million-Key Question—Investigating the Origins of RSA Public Keys.....	893
Petr Švenda, Matúš Nemec, Peter Sekan, Rudolf Kvašňovský, David Formánek, David Komárek, and Vashek Matyáš, <i>Masaryk University</i>	
Fingerprinting Electronic Control Units for Vehicle Intrusion Detection	911
Kyong-Tak Cho and Kang G. Shin, <i>University of Michigan</i>	
Lock It and Still Lose It—On the (In)Security of Automotive Remote Keyless Entry Systems	929
Flavio D. Garcia and David Oswald, <i>University of Birmingham</i> ; Timo Kasper, <i>Kasper & Oswald GmbH</i> ; Pierre Pavlidès, <i>University of Birmingham</i>	

Distributed Systems

OBLIVP2P: An Oblivious Peer-to-Peer Content Sharing System	945
Yaoqi Jia, <i>National University of Singapore</i> ; Tarik Moataz, <i>Colorado State University and Telecom Bretagne</i> ; Shruti Tople and Prateek Saxena, <i>National University of Singapore</i>	
AuthLoop: End-to-End Cryptographic Authentication for Telephony over Voice Channels	963
Bradley Reaves, Logan Blue, and Patrick Traynor, <i>University of Florida</i>	
You Are Who You Know and How You Behave: Attribute Inference Attacks via Users' Social Friends and Behaviors	979
Neil Zhenqiang Gong, <i>Iowa State University</i> ; Bin Liu, <i>Rutgers University</i>	

Web Measurements

Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016	997
Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner, <i>University of Washington</i>	

Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification	1015
Ben Stock, Giancarlo Pellegrino, and Christian Rossow, <i>Saarland University; Martin Johns, SAP SE; Michael Backes, Saarland University and Max Planck Institute for Software Systems (MPI-SWS)</i>	
You've Got Vulnerability: Exploring Effective Vulnerability Notifications	1033
Frank Li, <i>University of California, Berkeley</i> ; Zakir Durumeric, <i>University of Michigan, University of Illinois at Urbana–Champaign, and International Computer Science Institute</i> ; Jakub Czyz, <i>University of Michigan</i> ; Mohammad Karami, <i>George Mason University</i> ; Michael Bailey, <i>University of Illinois at Urbana–Champaign</i> ; Damon McCoy, <i>New York University</i> ; Stefan Savage, <i>University of California, San Diego</i> ; Vern Paxson, <i>University of California, Berkeley, and International Computer Science Institute</i>	

Proofs

Mirror: Enabling Proofs of Data Replication and Retrievability in the Cloud	1051
Frederik Armknecht, <i>University of Mannheim</i> ; Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame, <i>NEC Laboratories Europe</i>	
ZKBoo: Faster Zero-Knowledge for Boolean Circuits	1069
Irene Giacomelli, Jesper Madsen, and Claudio Orlandi, <i>Aarhus University</i>	
The Cut-and-Choose Game and Its Application to Cryptographic Protocols	1085
Ruiyu Zhu and Yan Huang, <i>Indiana University</i> ; Jonathan Katz, <i>University of Maryland</i> ; Abhi Shelat, <i>Northeastern University</i>	

Android

On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis.....	1101
Michael Backes, <i>Saarland University and Max Planck Institute for Software Systems (MPI-SWS)</i> ; Sven Bugiel and Erik Derr, <i>Saarland University</i> ; Patrick McDaniel, <i>The Pennsylvania State University</i> ; Damien Ochteau, <i>The Pennsylvania State University and University of Wisconsin—Madison</i> ; Sebastian Weisgerber, <i>Saarland University</i>	
Practical DIFC Enforcement on Android	1119
Adwait Nadkarni, Benjamin Andow, and William Enck, <i>North Carolina State University</i> ; Somesh Jha, <i>University of Wisconsin—Madison</i>	
Screen after Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images.....	1137
Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, and Dongyan Xu, <i>Purdue University</i> ; Golden G. Richard III, <i>University of New Orleans</i>	
Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis.....	1153
Yousra Aafer, Xiao Zhang, and Wenliang Du, <i>Syracuse University</i>	

Privacy

Identifying and Characterizing Sybils in the Tor Network	1169
Philipp Winter, <i>Princeton University and Karlstad University</i> ; Roya Ensafi, <i>Princeton University</i> ; Karsten Loesing, <i>The Tor Project</i> ; Nick Feamster, <i>Princeton University</i>	
k-fingerprinting: A Robust Scalable Website Fingerprinting Technique	1187
Jamie Hayes and George Danezis, <i>University College London</i>	
Protecting Privacy of BLE Device Users	1205
Kassem Fawaz, <i>University of Michigan</i> ; Kyu-Han Kim, <i>Hewlett Packard Labs</i> ; Kang G. Shin, <i>University of Michigan</i>	
Privacy in Epigenetics: Temporal Linkability of MicroRNA Expression Profiles	1223
Michael Backes, <i>Saarland University and Max Planck Institute for Software Systems (MPI-SWS)</i> ; Pascal Berrang, Anna Hecksteden, Mathias Humbert, Andreas Keller, and Tim Meyer, <i>Saarland University</i>	

Message from the 25th USENIX Security Symposium Program Co-Chairs

It is our pleasure to welcome you to the 25th USENIX Security Symposium in Austin, TX!

We hope you enjoy the outstanding program, which includes a mix of papers, invited talks, fun evening events, and, of course, the “hallway track.” Now in its 25th year, USENIX Security brings together researchers from both academia and industry interested in the latest advances in the security of computer systems and networks. The symposium is a premier venue for security and privacy research, and we look forward to seeing the lasting impact that this year’s papers will have in years to come.

For the first time, USENIX Security is chaired by two people. Given the growth of the symposium in the past few years and the huge number of submitted papers, this change was necessary. It was our great pleasure to chair USENIX Security this year, and now we want to take this opportunity to describe the process of creating the program you will enjoy over the next three days. This entire process was supported by 78 program committee members: 37 volunteers served as attending PC members, while 41 served as remote PC members. The PC spent countless hours not only reviewing papers but also discussing papers with each other online and in person. In total, more than 1,400 reviews and 2,300 comments were entered into the reviewing system, an average of five comments per paper. More specifically, the review process consisted of several rounds as described below.

First round of reviews (Feb. 25–March 24, 2016): We received 468 submissions, a 10% increase over last year. 12 papers were desk rejected due to a violation of submission requirements, and five papers were withdrawn by the authors after the deadline. The remaining 451 papers were assigned to at least two reviewers per submission. The program committee spent two weeks on online discussion once reviews had been collected. As in past years, we decided to finalize decisions in the first round for a subset of papers that had confident reviews and did not appear to have a chance of acceptance. Based on the positive feedback on early reject notifications, a feature introduced last year, we decided to also notify authors early about the status of their papers. All reviews from the first review round were sent out on April 7. In total, 233 papers were rejected in the first round of decisions and the remaining 218 papers moved on to the second review round.

This year, we introduced a new feature: authors had the option to appeal these initial reviews if they contained critical errors. We decided to introduce these appeals to make sure that we did not prematurely reject papers. An appeal was required to clearly and explicitly identify concrete disagreements with factual statements in the initial reviews. We received 19 appeals for papers rejected in the first round, and carefully checked each of them to understand the concerns raised by the authors. After several discussions, we decided to approve seven appeals and move these papers to the second review round. One of these papers was even finally accepted into the program, indicating that the intended process actually works. To our surprise, we also received 70 appeals for papers that were not rejected in the first round. It became clear that the intent of appeals was not clear to many authors, and we decided to make these appeals available to the reviewers so that they could be taken into account during the next phase.

Second round of reviews (April 6–May 3, 2016): In total, 225 papers were reviewed in the second round. Most papers received at least two more reviews, and controversial papers with diverging reviews were assigned at least three more reviews. After the second reviewing deadline had passed, the program committee spent an additional week discussing the papers using HotCRP. Each paper was discussed with the goal of reaching a consensus among the reviewers if the paper had a chance of acceptance into the final program.

Un-blinding papers (May 11, 2016): Outcomes and discussion points were finalized for each paper, and we as the PC co-chairs decided on the list of 88 papers to discuss at the PC meeting based on the recommendations. We arranged these papers into groups with a similar research topic so we could discuss them in batches during the PC meeting. Furthermore, we decided to pre-accept five papers, given that these papers had very strong positive reviews. Before the PC meeting, the author names were made visible to reviewers. The un-blinding was helpful during the meeting to clarify conflicts and to help prevent authors from being punished for failing to cite their own work or from reviewers who might have a bias based on a false assumption regarding the authors’ identity.

PC meeting (May 12–13, 2016, at Google in Mountain View, CA): 39 PC members attended the PC meeting. We allocated six minutes for the discussion of each paper. After going through the list of 88 papers, the PC spent several extra hours discussing tabled papers and papers that were voted to be resurrected. After the final decisions were made, we had accepted 72 papers, 15.4% of the submissions. The quality of these papers is very high—a testimony to the strength of our community!

The technical program would not have been possible without contributions from the program committee members and roughly 50 external reviewers who provided thoughtful reviews and recommendations. Please join us in thanking them for their countless hours of work! We would also like to thank Adrienne Porter-Felt for chairing the invited talks committee; Raluca Popa for serving as the poster session chair; Patrick Traynor for serving as the WiPs chair; student volunteer Paul Pearce for scribing at the PC meeting; Google for sponsoring the PC meeting; the USENIX staff, especially Casey Henderson and Michele Nelson, for all the support throughout the process; and the authors of all 468 papers for submitting their research for consideration. Finally, we would like to thank the USENIX steering committee for allowing us to have this incredible opportunity to work with so many wonderful people.

Thorsten Holz, *Ruhr-Universität Bochum*
Stefan Savage, *University of California, San Diego*
USENIX Security '16 Program Co-Chairs

Flip Feng Shui: Hammering a Needle in the Software Stack

Kaveh Razavi*
*Vrije Universiteit
Amsterdam*

Ben Gras*
*Vrije Universiteit
Amsterdam*

Cristiano Giuffrida
*Vrije Universiteit
Amsterdam*

Erik Bosman
*Vrije Universiteit
Amsterdam*

Bart Preneel
*Katholieke Universiteit
Leuven*

Herbert Bos
*Vrije Universiteit
Amsterdam*

* Equal contribution joint first authors

Abstract

We introduce Flip Feng Shui (FFS), a new exploitation vector which allows an attacker to induce bit flips over *arbitrary* physical memory in a *fully controlled way*. FFS relies on hardware bugs to induce bit flips over memory and on the ability to surgically control the physical memory layout to corrupt attacker-targeted data anywhere in the software stack. We show FFS is possible today with very few constraints on the target data, by implementing an instance using the *Rowhammer bug* and *memory deduplication* (an OS feature widely deployed in production). Memory deduplication allows an attacker to reverse-map any physical page into a virtual page she owns as long as the page’s contents are known. Rowhammer, in turn, allows an attacker to flip bits in controlled (initially unknown) locations in the target page.

We show FFS is extremely powerful: a malicious VM in a practical cloud setting can gain unauthorized access to a co-hosted victim VM running OpenSSH. Using FFS, we exemplify end-to-end attacks breaking OpenSSH public-key authentication, and forging GPG signatures from trusted keys, thereby compromising the Ubuntu/Debian update mechanism. We conclude by discussing mitigations and future directions for FFS attacks.

1 Introduction

The demand for high-performance and low-cost computing translates to increasing complexity in hardware and software. On the hardware side, the semiconductor industry packs more and more transistors into chips that serve as a foundation for our modern computing infrastructure. On the software side, modern operating systems are packed with complex features to support efficient resource management in cloud and other performance-sensitive settings.

Both trends come at the price of reliability and, inevitably, security. On the hardware side, components

are increasingly prone to failures. For example, a large fraction of the DRAM chips produced in recent years are prone to bit flips [34, 51], and hardware errors in CPUs are expected to become mainstream in the near future [10, 16, 37, 53]. On the software side, widespread features such as memory or storage deduplication may serve as side channels for attackers [8, 12, 31]. Recent work analyzes some of the security implications of both trends, but so far the attacks that abuse these hardware/software features have been fairly limited—probabilistic privilege escalation [51], in-browser exploitation [12, 30], and selective information disclosure [8, 12, 31].

In this paper, we show that an attacker abusing modern hardware/software properties can mount much more sophisticated and powerful attacks than previously believed possible. We describe Flip Feng Shui (FFS), a new exploitation vector that allows an attacker to induce bit flips over *arbitrary* physical memory in a *fully controlled way*. FFS relies on two underlying primitives: (i) the ability to induce bit flips in controlled (but not predetermined) physical memory pages; (ii) the ability to control the physical memory layout to reverse-map a target physical page into a virtual memory address under attacker control. While we believe the general vector will be increasingly common and relevant in the future, we show that an instance of FFS, which we term dFFS (i.e., deduplication-based FFS), can already be implemented on today’s hardware/software platforms with very few constraints. In particular, we show that by abusing Linux’ memory deduplication system (KSM) [6] which is very popular in production clouds [8], and the widespread Rowhammer DRAM bug [34], an attacker can *reliably* flip a single bit in *any* physical page in the software stack with known contents.

Despite the complete absence of software vulnerabilities, we show that a practical Flip Feng Shui attack can have devastating consequences in a common cloud setting. An attacker controlling a cloud VM can abuse

memory deduplication to seize control of a target physical page in a co-hosted victim VM and then exploit the Rowhammer bug to flip a particular bit in the target page in a fully controlled and reliable way without writing to that bit. We use dFFS to mount end-to-end corruption attacks against OpenSSH public keys, and Debian/Ubuntu update URLs and trusted public keys, all residing within the page cache of the victim VM. We find that, while dFFS is surprisingly practical and effective, existing cryptographic software is wholly unequipped to counter it, given that “*bit flipping is not part of their threat model*”. Our end-to-end attacks completely compromise widespread cryptographic primitives, allowing an attacker to gain full control over the victim VM.

Summarizing, we make the following contributions:

- We present FFS, a new exploitation vector to induce hardware bit flips over arbitrary physical memory in a controlled fashion (Section 2).
- We present dFFS, an implementation instance of FFS that exploits KSM and the Rowhammer bug and we use it to bit-flip RSA public keys (Section 3) and compromise authentication and update systems of a co-hosted victim VM, granting the attacker unauthorized access and privileged code execution (Section 4).
- We use dFFS to evaluate the time requirements and success rates of our proposed attacks (Section 5) and discuss mitigations (Section 6).

The videos demonstrating dFFS attacks can be found in the following URL:

<https://vusec.net/projects/flip-feng-shui>

2 Flip Feng Shui

To implement an FFS attack, an attacker requires a *physical memory massaging primitive* and a *hardware vulnerability* that allows her to flip bits on certain locations on the medium that stores the users’ data. Physical memory massaging is analogous to virtual memory massaging where attackers bring the virtual memory into an exploitable state [23, 24, 55], but instead performed on physical memory. Physical memory massaging (or simply *memory massaging*, hereafter) allows the attacker to steer victim’s sensitive data towards those physical memory locations that are amenable to bit flips. Once the target data land on the intended vulnerable locations, the attacker can trigger the hardware vulnerability and corrupt the data via a controlled bit flip. The end-to-end attack allows the attacker to flip *a bit of choice in data of choice anywhere* in the software stack in a controlled fashion.

With some constraints, this is similar to a typical arbitrary memory write primitive used for software exploitation [15], with two key differences: (i) the end-to-end attack requires no software vulnerability; (ii) the attacker can overwrite arbitrary physical (not just virtual) memory on the running system. In effect, FFS transforms an underlying hardware vulnerability into a very powerful software-like vulnerability via three fundamental steps:

1. *Memory templating*: identifying physical memory locations in which an attacker can induce a bit flip using a given hardware vulnerability.
2. *Memory massaging*: steering targeted sensitive data towards the vulnerable physical memory locations.
3. *Exploitation*: triggering the hardware vulnerability to corrupt the intended data for exploitation.

In the remainder of this section, we detail each of these steps and outline FFS’s end-to-end attack strategy.

2.1 Memory Templating

The goal of the memory templating step is to fingerprint the hardware bit-flip patterns on the running system. This is necessary, since the locations of hardware bit flips are generally unknown in advance. This is specifically true in the case of Rowhammer; every (vulnerable) DRAM module is unique in terms of physical memory offsets with bit flips. In this step, the attacker triggers the hardware-specific vulnerability to determine which physical pages, and which offsets within those pages are vulnerable to bit flips. We call the combination of a vulnerable page and the offset a *template*.

Probing for templates provides the attacker with knowledge of *usable* bit flips. Thanks to Flip Feng Shui, any template can potentially allow the attacker to exploit the hardware vulnerability over physical memory in a controlled way. The usefulness of such an exploit, however, depends on the direction of the bit flip (i.e., one-to-zero or zero-to-one), the page offset, and the contents of the target victim page. For each available template, the attacker can only craft a Flip Feng Shui primitive that corrupts the target data page with the given *flip* and *offset*. Hence, to surgically target the victim’s sensitive data of interest, the attacker needs to probe for matching templates by repeatedly exploiting the hardware vulnerability over a controlled physical page (i.e., mapped in her virtual address space). To perform this step efficiently, our own dFFS implementation relies on a variant of double-sided Rowhammer [51]. Rowhammer allows an attacker to induce bit flips in vulnerable memory locations by repeatedly reading from memory pages located in adjacent rows. We discuss the low-level details

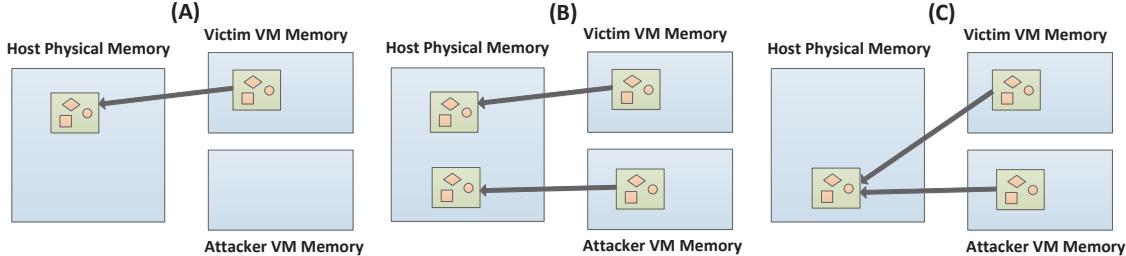


Figure 1: Memory deduplication can provide an attacker control over the layout of physical memory.

of the Rowhammer vulnerability and our implementation in Section 4.2.

2.2 Memory Massaging

To achieve bit flips over arbitrary contents of the victim’s physical memory, FFS abuses modern memory management patterns and features to craft a memory massaging primitive. Memory massaging allows the attacker to map a desired victim’s physical memory page into her own virtual memory address space in a controllable way.

Given a set of templates and the memory massaging primitive, an ideal version of FFS can corrupt any of the victim’s memory pages at an offset determined by the selected template.

While memory massaging may be nontrivial in the general case, it is surprisingly easy to abuse widely deployed memory deduplication features to craft practical FFS attacks that corrupt any of the victim’s memory pages with *known contents* (similar to our dFFS implementation). Intuitively, since memory deduplication merges system-wide physical memory pages with the same contents, an attacker able to craft the contents of any of the victim’s memory pages can obtain a memory massaging primitive and map the target page into her address space.

Figure 1 shows how an attacker can control the physical memory location of a victim VM’s memory page. At first, the attacker needs to predict the contents of the victim VM’s page that she wants to control (Figure 1-A). Once the target page is identified, the attacker VM creates a memory page with the same contents as the victim VM’s memory page and waits for the memory deduplication system to scan both pages (Figure 1-B). Once the two physical pages (i.e., the attacker’s and the victim’s pages) are identified, the memory deduplication system returns one of the two pages back to the system, and the other physical page is used to back both the attacker and the victim’s (virtual) pages. If the attacker’s page is used to back the memory of the victim page, then, in effect, the attacker controls the physical memory location of the victim page (Figure 1-C).

There are additional details necessary to craft a memory massaging primitive using a real-world implementation of memory deduplication (e.g., KSM). Section 4.1 elaborates on such details and presents our implementation of memory massaging on Linux.

2.3 Exploitation

At this stage, FFS already provides the attacker with templated bit flips over the victim’s physical memory pages with known (or predictable) contents. The exploitation surface is only subject to the available templates and their ability to reach interesting locations for the attacker. As we will see, the options are abundant.

While corrupting the memory state of running software of the victim is certainly possible, we have opted for a more straightforward, yet extremely powerful exploitation strategy. We consider an attacker running in a cloud VM and seeking to corrupt interesting contents in the page cache of a co-hosted victim VM. In particular, our dFFS implementation includes two exploits that corrupt sensitive file contents in the page cache in complete absence of software vulnerabilities:

1. Flipping SSH’s `authorized_keys`: assuming the RSA public keys of the individuals accessing the victim VM are known, an attacker can use dFFS to induce an exploitable flip in their public keys, making them prone to factorization and breaking the authentication system.
2. Flipping apt’s `sources.list` and `trusted.gpg`: Debian/Ubuntu’s apt package management system relies on the `sources.list` file to operate daily updates and on the `trusted.gpg` file to check the authenticity of the updates via RSA public keys. Compromising these files allows an attacker to make a victim VM download and install arbitrary attacker-generated packages.

In preliminary experiments, we also attempted to craft an exploit to bit-flip SSH’s `moduli` file containing Diffie-Hellman group parameters and eavesdrop on the victim

VM’s SSH traffic. The maximum group size on current distributions of OpenSSH is 1536. When we realized that an exploit targeting such 1536-bit parameters would require a nontrivial computational effort (see Appendix A for a formal analysis), we turned our attention to the two more practical and powerful exploits above.

In Section 3, we present a cryptanalysis of RSA moduli with a bit flip as a result of our attacks. In Section 4, we elaborate on the internals of the exploits, and finally, in Section 5, we evaluate their success rate and time requirements in a typical cloud setting.

3 Cryptanalysis of RSA with Bit Flips

RSA [49] is a public-key cryptosystem: the sender encrypts the message with the public key of the recipient (consisting of an exponent e and a modulus n) and the recipient decrypts the ciphertext with her private key (consisting of an exponent d and a modulus n). This way RSA can solve the key distribution problem that is inherent to symmetric encryption. RSA can also be used to digitally sign messages for data or user authentication: the signing operation is performed using the private key, while the verification operation employs the public key.

Public-key cryptography relies on the assumption that it is computationally infeasible to derive the private key from the public key. For RSA, computing the private exponent d from the public exponent e is believed to require the factorization of the modulus n . If n is the product of two large primes of approximately the same size, factorizing n is not feasible. Common sizes for n today are 1024 to 2048 bits.

In this paper we implement a fault attack on the modulus n of the victim: we corrupt a single bit of n , resulting in n' . We show that with high probability n' will be easy to factorize. We can then compute from e the corresponding value of d' , the private key, that allows us to forge signatures or to decrypt. We provide a detailed analysis of the expected computational complexity of factorizing n' in the following¹.

RSA performs computations modulo n , where t is the bitlength of n ($t = 1 + \lfloor \log_2 n \rfloor$). Typical values of t lie between 512 (export control) and 8192, with 1024 and 2048 the most common values. We denote the i th bit of n with $n[i]$ ($0 \leq i < t$), with the least significant bit (LSB) corresponding to $n[0]$. The unit vector is written as e_i , that is $e_i[i] = 1$ and $e_i[j] = 0$, for $j \neq i$. The operation of flipping the i th bit of n results in n' , or $n' = n \oplus e_i$. Any integer can be written as the product of primes, hence $n = \prod_{i=1}^s p_i^{\gamma_i}$, where p_i are the prime factors of n , γ_i is the multiplicity of p_i and s is the number of distinct prime

¹A similar analysis for Diffie-Hellman group parameters with bit flips can be found in Appendix A.

factors. W.l.o.g. we assume that $p_1 > p_2 > \dots > p_s$.

In the RSA cryptosystem, the modulus n is the product of two odd primes p_1, p_2 of approximate equal size, hence $s = 2$, and $\gamma_1 = \gamma_2 = 1$. The encryption operation is computed as $c = m^e \bmod n$, with e the public exponent, and $m, c \in [0, n - 1]$ the plaintext respectively the ciphertext. The private exponent d can be computed as $d = e^{-1} \bmod \lambda(n)$, with $\lambda(n)$ the Carmichael function, given by $\text{lcm}(p_1, p_2)$. The best known algorithm to recover the private key is to factorize n using the General Number Field Sieve (GNFS) (see e.g. [42]), which has complexity $O(L_n[1/3, 1.92])$, with

$$L_n[a, b] = \exp((b + o(1))(\ln n)^a (\ln \ln n)^{1-a}).$$

For a 512-bit modulus n , Adrian et al. estimate that the cost is about 1 core-year [3]. The current record is 768 bits [35], but it is clear that 1024 bits is within reach of intelligence agencies [3].

If we flip the LSB of n , we obtain $n' = n - 1$, which is even hence $n' = 2 \cdot n''$ with n'' a $t - 1$ -bit integer. If we flip the most significant bit of n , we obtain the odd $t - 1$ -bit integer n' . In all the other cases we obtain an odd t -bit integer n' . We conjecture that the integer n'' (for the LSB case) and the integers n' (for the other cases) have the same distribution of prime factors as a random odd integer. To simplify the notation, we omit in the following the LSB case, but the equations apply with n' replaced by n'' .

Assume that an attacker can introduce a bit flip to change n into n' with as factorization $n = \prod_{j=1}^{t'} p_i'^{\tilde{\gamma}_i}$. Then $c' = m'^e \bmod n'$. The Carmichael function can be computed as

$$\lambda(n') = \text{lcm}\left(\left\{p_i'^{\tilde{\gamma}_i-1} \cdot (p_i' - 1)\right\}\right).$$

If $\gcd(e, \lambda(n')) = 1$, the private exponent d' can be found as $d' = e^{-1} \bmod \lambda(n')$. For prime exponents e , the probability that $\gcd(e, \lambda(n')) > 1$ equals $1/e$. For $e = 3$, this means that 1 in 3 attacks fails, but for the widely used value $e = 2^{16} + 1$, this is not a concern. With the private exponent d' we can decrypt or sign any message. Hence the question remains how to factorize n' . As it is very likely that n' is not the product of two primes of almost equal size, we can expect that factorizing n' is much easier than factorizing n .

Our conjecture implies that with probability $2/\ln n'$, n' is prime and in that case the factorization is trivial. If n' is composite, the best approach is to find small factors (say up to 16 bits) using a greatest common divisor operation with the product of the first primes. The next step is to use Pollard’s ρ algorithm (or Brent’s variant) [42]: this algorithm can easily find factors up to 40...60 bits. A third step consists of Lenstra’s Elliptic Curve factorization Method (ECM) [38]: ECM can quickly find factors up to 60...128 bits (the record is a factor of about

270 bits²). Its complexity to find the smallest prime factor p'_s is equal to $O(L_{p'_s}[1/2, \sqrt{2}])$. While ECM is asymptotically less efficient than GNFS (because of the parameter 1/2 rather than 1/3), the complexity of ECM depends on the size of the smallest prime factor p'_s rather than on the size of the integer n' to factorize. Once a prime factor p'_i is found, n' is divided by it, the result is tested for primality and if the result is composite, ECM is restarted with as argument n'/p'_i .

The complexity analysis of ECM depends on the number of prime factors and the distribution of the size of the second largest prime factor p'_2 : it is known that its expected value is $0.210 \cdot t$ [36]. The Erdős–Kac theorem [22] states that the number $\omega(n')$ of distinct prime factors of n' is normally distributed with mean and variance $\ln \ln n'$: for $t = 1024$ the mean is about 6.56, with standard deviation 2.56. Hence it is unlikely that we have exactly two prime factors (probability 3.5%), and even less likely that they are of approximate equal size. The probability that n' is prime is equal to 0.28%. The expected size of the second largest prime factor p'_2 is 215 bits and the probability that it has less than 128 bits is 0.26 [36]. In this case ECM should be very efficient. For $t = 2048$, the probability that n' is prime equals 0.14%. The expected size of the second largest prime factor p'_2 is 430 bits; the probability that p'_2 has less than 228 bits is 0.22 and the probability that it has less than 128 bits is about 0.12. Similarly, for $t = 4096$, the expected size of the second largest prime factor p'_2 is 860 bits. The probability that p'_2 has less than 455 bits is 0.22.

The main conclusion is that *if n has 1024-2048 bits, we can expect to factorize n' efficiently with a probability of 12 – 22% for an arbitrary bit flip, but larger moduli should also be feasible*. As we show in Section 5, given a few dozen templates, we can easily factorize any 1024 bit to 4096 bit modulus with one (or more) of the available templates.

4 Implementation

To implement dFFS reliably on Linux, we need to understand the internals of two kernel subsystems, kernel same-page merging [6] (KSM) and transparent huge pages [5], and the way they interact with each other. After discussing them and our implementation of the Rowhammer exploit (Sections 4.1, 4.2, and 4.3), we show how we factorized corrupted RSA moduli in Section 4.4 before summarizing our end-to-end attacks in Section 4.5.

²https://en.wikipedia.org/wiki/Lenstra_elliptic_curve_factorization

4.1 Kernel Same-page Merging

KSM, the Linux implementation of memory deduplication, uses a kernel thread that periodically scans memory to find memory pages with the same contents that are candidates for merging. It then keeps a single physical copy of a set of candidate pages, marks it read-only, and updates the page-table entries of all the other copies to point to it before releasing their physical pages to the system.

KSM keeps two red-black trees, termed “stable” and “unstable”, to keep track of the merged and candidate pages. The merged pages reside in the stable tree while the candidate contents that are not yet merged are in the unstable tree. KSM keeps a list of memory areas that are registered for deduplication and goes through the pages in these areas in the order in which they were registered. For each page that it scans, it checks if the stable tree already contains a page with the same contents. If so, it updates the page-table entry for that page to have it point to the physical page in the stable tree and releases the backing physical page to the system. Otherwise, it searches the unstable tree for a match and if it finds one, promotes the page to the stable tree and updates *the page-table entry of the match to make it point to this page*. If no match is found in either one of the trees, the page is added to the unstable tree. After going through all memory areas, KSM dumps the unstable tree before starting again. Further details on the internals of KSM can be found in [6].

In the current implementation of KSM, during a merge, the physical page in either the stable tree or the unstable tree is always preferred. This means that during a merge with a page in the stable tree, the physical location of the page in the stable tree is chosen. Similarly, the physical memory of the page in the unstable tree is chosen to back both pages. KSM scans the memory of the VMs in order that they have been registered (i.e., their starting time). This means that to control the location of the target data on physical memory using the *unstable tree* the attacker VM should have been started *before* the victim VM. Hence, the longer the attacker VM waits, the larger the chance of physical memory massaging through the unstable tree.

The better physical memory massaging possibility is through the stable tree. An attacker VM can upgrade a desired physical memory location to the stable tree by creating two copies of the target data and placing one copy in the desired physical memory location and another copy in a different memory location. By ensuring that the other copy comes after the desired physical memory location in the physical address-space, KSM merges the two pages and creates a stable tree node using the desired physical memory location. At this point, any other



Figure 2: A SO-DIMM with its memory chips.

page with the same contents will assume the same physical memory location desired by the attacker VM. For this to work, however, the attacker needs to control when the memory page with the target contents is created in the victim VM. In the case of our OpenSSH attack, for example, the attacker can control when the target page is created in the victim VM by starting an SSH connection using an invalid key with the target username.

For simplicity, the current version of dFFS implements the memory massaging using the unstable tree by assuming that the attacker VM has started first, but it is trivial to add support for memory massaging with stable tree. Using either the stable or unstable KSM trees for memory massaging, all dFFS needs to do is crafting a page with the same contents as the victim page and place it at the desired physical memory page; KSM will then perform the necessary page-table updates on dFFS’s behalf! In other words, KSM inadvertently provides us with exactly the kind of memory massaging we need for successful Flip Feng Shui.

4.2 Rowhammer inside KVM

Internally, DRAM is organized in rows. Each row provides a number of physical cells that store memory bits. For example, in an x86 machine with a single DIMM, each row contains 1,048,576 cells that can store 128 kB of data. Each row is internally mapped to a number of chips on the DIMM as shown in Figure 2.

Figure 3 shows a simple organization of a DRAM chip. When the processor reads a physical memory location, the address is translated to an offset on row i of the DRAM. Depending on the offset, the DRAM selects the proper chip. The selected chip then copies the contents of its row i to the row buffer. The contents at the correct offset within the row buffer is then sent on the bus to the processor. The row buffer acts as a cache: if the selected row is already in the row buffer, there is no need to read from the row again.

Each DRAM cell is built using a transistor and a capacitor. The transistor controls whether the contents of the cell is accessible, while the capacitor can hold a charge which signifies whether the stored content is a

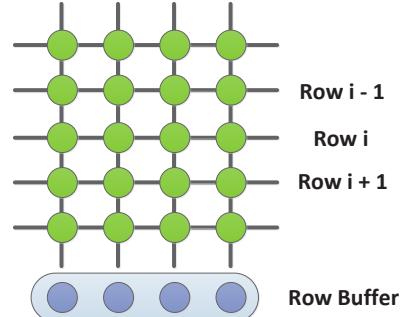


Figure 3: DRAM’s internal organization.

high or low bit. Since capacitors leak charge over time, the processor sends refresh commands to DIMM rows in order to recharge their contents. On top of the refresh commands, every time a row is read by the processor, the chip also recharges its cells.

As DRAM components have become smaller, they keep a smaller charge to signify stored contents. With a smaller charge, the error margin for identifying whether the capacitor is charged (i.e., the stored value) is also smaller. Kim et al. [34] showed that the smaller error margin, in combination with unexpected charge exchange between cells of different rows, can result in the cell to “lose” its content. To trigger this DRAM reliability issue, an attacker needs fast activations of DRAM rows which causes a cell in adjacent rows to lose enough charge so that its content is cleared. Note that due to the row buffer, at least two rows need to activate one after the other in a tight loop for Rowhammer to trigger. If only one row is read from, the reads can be satisfied continually from the row buffer, without affecting the row charges in the DRAM cells.

Double-sided Rowhammer. Previous work [51] reported that if these two “aggressor” rows are selected in a way that they are one row apart (e.g., row $i - 1$ and $i + 1$ in Figure 3), the chances of charge interaction between these rows and the row in the middle (i.e., row i) increases, resulting in potential bit flips in that row. This variant of Rowhammer is named double-sided Rowhammer. Apart from additional speed for achieving bit flips, it provides additional reliability by isolating the location of most bit flips to a certain (victim) row.

To perform double-sided Rowhammer inside KVM, we need to know the host physical addresses inside the VM. This information is, however, not available in the guest: guest physical addresses are mapped to host virtual addresses which can be mapped to any physical page by the Linux kernel. Similar to [30], we rely on transparent huge pages [5] (THP). THP is a Linux kernel feature

that runs in the background and merges virtually contiguous normal pages (i.e., 4 kB pages) into huge pages (i.e., 2 MB pages) that rely on contiguous pieces of physical memory. THP greatly reduces the number of page-table entries in suitable processes, resulting in fewer TLB³ entries. This improves performance for some workloads.

THP is another (weak) form of memory massaging; it transparently allows the attacker control over how the system maps guest physical memory to host physical memory. Once the VM is started and a certain amount of time has passed, THP will transform most of the VM’s memory into huge pages. Our current implementation of dFFS runs entirely in the userspace of the guest and relies on the default-on THP feature of both the host and the guest. As soon as the guest boots, dFFS allocates a large buffer with (almost) the same size as the available memory in the guest. The THP in the host then converts guest physical addresses into huge pages and the THP in the guest turns the guest virtual pages backing dFFS’s buffer into huge pages as well. As a result, dFFS’s buffer will largely be backed by huge pages all the way down to host physical memory.

To make sure that the dFFS’s buffer is backed by huge pages, we request the guest kernel to align the buffer at a 2 MB boundary. This ensures that if the buffer is backed by huge pages, it starts with one: on the x86_64 architecture, the virtual and physical huge pages share the lowest 20 bits, which are zero. The same applies when transitioning from the guest physical addresses to host physical addresses. With this knowledge, dFFS can assume that the start of the allocated buffer is the start of a memory row, and since multiple rows fit into a huge page, it can successively perform double-sided Rowhammer on these rows. To speed up our search for bit flips during double-sided Rowhammer on each two rows, we rely on the row-conflict side channel for picking the hammering addresses within each row [44]. We further employed multiple threads to amplify the Rowhammer effect.

While THP provides us with the ability to efficiently and reliably induce Rowhammer bit flips, it has unexpected interactions with KSM that we will explore in the next section.

4.3 Memory Massaging with KSM

In Section 2.2, we discussed the operational semantics of KSM. Here we detail some of its implementation features that are important for dFFS.

³TLB or translation lookaside buffer is a general term for processor caches for page-table entries to speed up the virtual to physical memory translation

4.3.1 Interaction with THP

As we discussed earlier, KSM deduplicates memory pages with the same contents as soon as it finds them. KSM currently does not support deduplication of huge pages, but what happens when KSM finds matching contents within huge pages?

A careful study of the KSM shows that KSM always prefers reducing memory footprint over reducing TLB entries; that is, KSM breaks down huge pages into smaller pages if there is a small page inside with similar contents to another page.

This specific feature is important for an efficient and reliable implementation of dFFS, but has to be treated with care. More specifically, we can use huge pages as we discussed in the previous section for efficient and reliable double-sided Rowhammer, while retaining control over which victim page we should map in the middle of our target (vulnerable) huge page.

KSM, however, can have undesired interactions with THP from dFFS’s point of view. If KSM finds pages in the attacker VM’s memory that have matching contents, it merges them with each other or with a page from a previously started VM. In these situations, KSM breaks THP by releasing one of its smaller pages to the system. To avoid this, dFFS uses a technique to avoid KSM during its templating phase. KSM takes a few tens of seconds to mark the pages of dFFS’s VM as candidates for deduplication. This gives dFFS enough time to allocate a large buffer with the same size as VM’s available memory (as mentioned earlier) and write unique integers at a pre-determined location within each (small) page of this buffer as soon as its VM boots. The entropy present within dFFS’s pages then prohibits KSM to merge these pages which in turn avoids breaking THP.

4.3.2 On dFFS Chaining

Initially, we planned on chaining memory massaging primitive and FFS to induce an arbitrary number of bit flips at many desired locations of the victim’s memory page. After using the first template for the first bit flip, the attacker can write to the merged memory page to trigger a copy-on-write event that ultimately unmerges the two pages (i.e., the attacker page from the victim page). At this stage, the attacker can use dFFS again with a new template to induce another bit flip.

However, the implementation of KSM does not allow this to happen. During the copy-on-write event, the victim’s page remains in the stable tree, even if it is the only remaining page. This means that subsequent attempts for memory massaging results in the victim page to control the location of physical memory, disabling the attacker’s ability for chaining FFS attacks.

Even so, based on our single bit flip cryptanalysis on public keys and our evaluation in Section 5, chaining is not necessary for performing successful end-to-end attacks with dFFS.

4.4 Attacking Weakened RSA

For the two attacks in this paper, we generate RSA private keys, i.e., the private exponents d' corresponding to corrupted moduli n' (as described in Section 3). We use d' to compromise two applications: OpenSSH and GPG.

Although the specifics of the applications are very different, the pattern to demonstrate each attack is the same and as follows:

1. Obtain the file containing the RSA public key (n, e) . This is application-specific, but due to the nature of public key cryptosystems, generally unprotected. We call this the input file.
2. Using the memory templating step of Section 2.1 we obtain a list of templates that we are able to flip within a physical page. We flip bits according to the target templates to obtain corrupted keys. For every single bitflip, we save a new file. We call these files the corrupted files. According to the templating step, dFFS has the ability to create any of these corrupted files in the victim by flipping a bit in the page cache.
3. One by one, we now read the (corrupted) public keys for each corrupted file. If the corrupted file is parsed correctly *and* the public key has a changed modulus $n' \neq n$ and the same e , this n' is a candidate for factorization.
4. We start factorizations of all n' candidates found in the previous step. As we described in Section 3, the best known algorithm for our scenario is ECM that finds increasingly large factor in an iterative fashion. We use the Sage [19] implementation of ECM for factorizing n' . We invoke an instance of ECM per available core for each corrupted key with a 1 hour timeout (all available implementations of ECM run with a single thread).
5. For all successful factorizations, we compute the private exponent d' corresponding to (n', e) and generate the corresponding private key to the corrupted public key. How to compute d' based on the factorization of n' is described in Section 3. We can then use the private key with the unmodified application. This step is application-specific and we will discuss it for our case studies shortly.

We now describe our end-to-end attacks that put all the pieces of dFFS together using two target applications: OpenSSH and GPG.

4.5 End-to-end Attacks

Attacker model. The attacker owns a VM co-hosted with a victim VM on a host with DIMMs susceptible to Rowhammer. We further assume that memory deduplication is turned on—as is common practice in public cloud settings [8]. The attacker has the ability to use the memory deduplication side-channel to fingerprint low-entropy information, such as the IP address of the victim VM, OS/library versions, and the usernames on the system (e.g., through `/etc/passwd` file in the page cache) as shown by previous work [32, 43, 56]. The attacker’s goal is to compromise the victim VM without relying on any software vulnerability. We now describe how this model applies with dFFS in two important and widely popular applications.

4.5.1 OpenSSH

One of the most commonly used authentication mechanisms allowed by the OpenSSH daemon is an RSA public key. By adding a user’s RSA public key to the SSH `authorized_keys` file, the corresponding private key will allow login of that user without any other authentication (such as a password) in a default setting. The public key by default includes a 2048 bit modulus n . The complete key is a 372-byte long base64 encoding of (n, e) .

The attacker can initiate an SSH connection to the victim with a correct victim username and an arbitrary private key. This interaction forces OpenSSH to read the `authorized_keys` file, resulting in this file’s contents getting copied into the page cache at the right time as we discussed in Section 4.1. Public key cryptosystems by definition do not require public keys to be secret, therefore we assume an attacker can obtain a victim public key. For instance, GitHub makes the users’ submitted SSH public keys publicly available [27].

With the victim’s public key known and in the page cache, we can initiate dFFS for inducing a bit flip. We cannot flip just any bit in the memory page caching the `authorized_keys`; some templates *will* break the base64 encoding, resulting in a corrupted file that OpenSSH does not recognize. Some flips, however, decode to a valid (n', e) key that we can factorize. We report in Section 5 how many templates are available on average for a target public key.

Next, we use a script with the PyCrypto RSA cryptographic library [39] to operate on the corrupted public keys. This library is able to read and parse OpenSSH public key files, and extract the RSA parameters (n, e) .

It can also generate RSA keys with specific parameters and export them as OpenSSH public (n', e) and private (n', d') keys again. All the attacker needs to do is factorize n' as we discussed in Section 4.4.

Once we know the factors of n' , we generate the private key (n', d') that can be used to login to the victim VM using an unmodified OpenSSH client.

4.5.2 GPG

The GNU Privacy Guard, or GPG, is a sophisticated implementation of, among others, the RSA cryptosystem. It has many applications in security, one of which is the verification of software distributions by verifying signatures using trusted public keys. This is the larger application we intend to subvert with this attack.

Specifically, we target the `apt` package distribution system employed by Debian and Ubuntu distribution for software installation and updates. `apt` verifies package signatures after download using GPG and trusted public keys stored in `trusted.gpg`. It fetches the package index from sources in `sources.list`.

Our attack first steers the victim to our malicious repository. The attacker can use dFFS to achieve this goal by inducing a bit flip in the `sources.list` file that is present in the page cache after an update. `sources.list` holds the URL of the repositories that are used for package installation and update. By using a correct template, the attacker can flip a bit that results in a URL that she controls. Now, the victim will seek the package index and packages at an attacker-controlled repository.

Next, we use our exploit to target the GPG trusted keys database. As this file is part of the software distribution, the stock contents of this file is well-known and we assume this file is unchanged or we can guess the new changes. (Only the pages containing the keys we depend on need be either unchanged or guessed.) This file resides in the page cache every time the system tries to update as a result of a daily cron job, so in this attack, no direct interaction with the victim is necessary for bringing the file in the page cache. Our implicit assumption is that this file remains in the page cache for the next update iteration.

Similar to OpenSSH, we apply bit flip mutations in locations where we can induce bit flips according to the memory templating step. As a result, we obtain the corrupted versions of this file, and each time check whether GPG will still accept this file as a valid keyring and that one of the RSA key moduli has changed as a result of our bit flip. Extracting the key data is done with the GPG `--list-keys --with-key-data` options.

For every bitflip location corresponding to a corrupted modulus that we can factorize, we pick one of these

mutations and generate the corresponding (n', d') RSA private key, again using PyCrypto. We export this private key using PyCrypto as PEM formatted key and use `pem2openpgp` [26] to convert this PEM private key to the GPG format. Here we specify the usage flags to include signing and the same generation timestamp as the original public key. We can then import this private key for use for signing using an unmodified GPG.

It is important that the Key ID in the private keyring match with the Key ID in the `trusted.gpg` file. This Key ID is not static but is based on a hash computed from the public key data, a key generation timestamp, and several other fields. In order for the Key ID in the private keyring to match with the Key ID in the public keyring, these fields have to be identical and so the setting of the creation timestamp is significant.

One significant remark about the Key ID changing (as a result of a bit flip) is that this caused the self-signature on the public keyring to be ignored by GPG! The signature contains the original Key ID, but it is now attached to a key with a different ID due to the public key mutation. As a result, *GPG ignores the attached signature as an integrity check of the bit-flipped public key and the self-signing mechanism fails to catch our bit flip*. The only side-effect is harmless to our attack – GPG reports that the trusted key is not signed. `apt` ignores this without even showing a warning. After factorizing the corrupted public key modulus, we successfully verified that the corresponding private key can generate a signature that verifies against the bit-flipped public key stored in the original `trusted.gpg`.

We can now sign our malicious package with the new private key and the victim will download and install the new package without a warning.

5 Evaluation

We evaluated dFFS to answer the following three key questions:

- What is the success probability of the dFFS attack?
- How long does the dFFS attack take?
- How much computation power is necessary for a successful dFFS attack?

We used the following methodology for our evaluation. We first used a Rowhammer testbed to measure how many templates are available in a given segment of memory and how long it takes us to find a certain template. We then executed the end-to-end attacks discussed in Section 4.5 and report on their success rate and their start-to-finish execution time. We then performed an analytical large-scale study of the factorization time, success probability, and computation requirements of 200

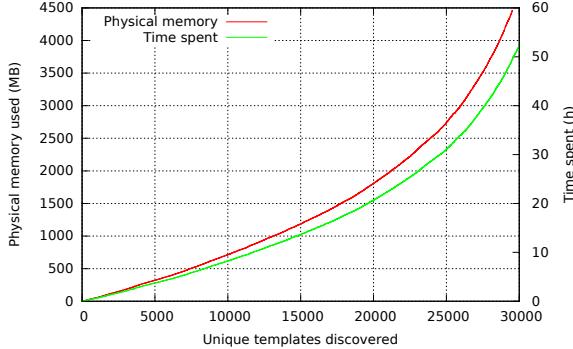


Figure 4: Required time and memory for templating.

RSA public keys for each of the 1024, 2048 and 4096-bit moduli with 50 bit flips at random locations (i.e., 30,000 bit flipped public keys in total).

We used the following hardware for our Rowhammer testbed and for the cluster that we used to conduct our factorization study:

Rowhammer testbed. Intel Haswell i7-4790 4-core processor on an Asus H97-Pro mainboard with 8 GB of DDR3 memory.

Factorization cluster. Up to 60 nodes, each with two Intel Xeon E5-2630 8-core processors with 64 GB of memory.

5.1 dFFS on the Rowhammer Testbed

Memory templating. Our current implementation of Rowhammer takes an average of 10.58 seconds to complete double-sided Rowhammer for each target row. Figure 4 shows the amount of time and physical memory that is necessary for discovering a certain number of templates. Note that, in our testbed, we could discover templates for almost any bit offset (i.e., 29,524 out of 32,768 possible templates). Later, we will show that we only need a very small fraction of these templates to successfully exploit our two target programs.

Memory massaging. dFFS needs to wait for a certain amount of time for KSM to merge memory pages. KSM scans a certain number of pages in each waking period. On the default version of Ubuntu, for example, KSM scans 100 pages every 20 milliseconds (i.e., 20 MB). Recent work [12] shows that it is possible to easily detect when a deduplication pass happens, hence dFFS needs to wait at most the sum of memory allocated to each co-hosted VM. For example, in our experiments with one

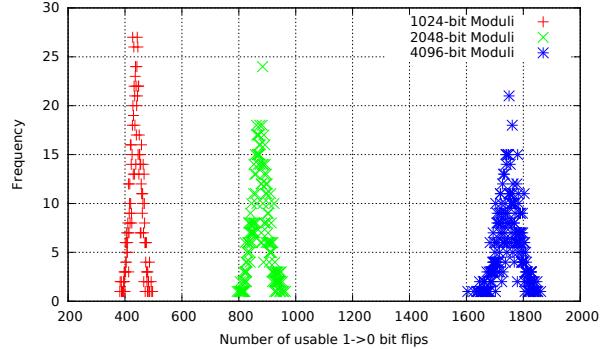


Figure 5: Number of usable 1-to-0 bit flips usable in the SSH authorized_keys file for various modulus sizes.

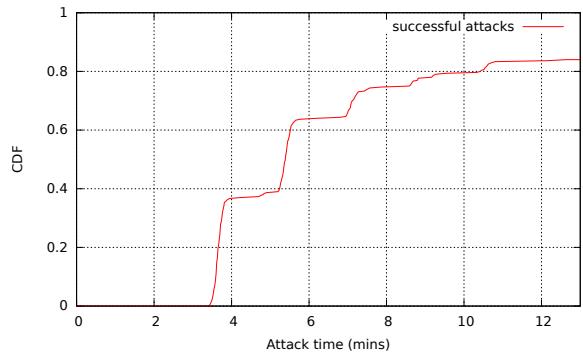


Figure 6: CDF of successful automatic SSH attacks.

attacker VM and one victim VM each with 2 GB of memory, KSM takes at most around 200 seconds for a complete pass.

5.2 The SSH Public Key Attack

Figure 5 shows the number of possible templates to perform the dFFS attack on the SSH authorized_keys file with a single randomly selected RSA public key, for 1024, 2048 and 4096-bit public keys. For this experiment, we assumed 1-to-0 bit flips since they are more common in our testbed. For DRAM chips that are susceptible to frequent 0-to-1 bit flips, these numbers double. For our experiment we focused on 2048-bit public keys as they are the default length as generated by the ssh-keygen command.

To demonstrate the working end-to-end attack, measure its reliability, and measure the elapsed time distribution, we automatically performed the SSH attack 300 times from an attacker VM on a victim VM, creating the keys and VM's from scratch each time. Figure 6 shows the CDF of successful attacks with respect to the time they took. In 29 cases (9.6%), the Rowhammer operation did not change the modulus at all (the attacker needs

Table 1: Examples of domains that are one bit flip away from `ubuntu.com` that we purchased.

<code>ubuftu.com</code>	<code>ubunt5.com</code>	<code>ubunte.com</code>
<code>ubunuu.com</code>	<code>ubunvu.com</code>	<code>ubunpu.com</code>
<code>ubun4u.com</code>	<code>ubuntw.com</code>	<code>ubuntt.com</code>

to retry). In 19 cases (6.3%), the Rowhammer operation changed the modulus other than planned. The remaining 252 (84.1%) were successful the first time. All the attacks finished within 12.6 minutes with a median of 5.3 minutes.

5.3 The Ubuntu/Debian Update Attack

We tried factorizing the two bit-flipped 4096 bit **Ubuntu Archive Automatic Signing** RSA keys found in the `trusted.gpg` file. Out of the 8,192 trials (we tried both 1-to-0 and 0-to-1 flips), we could factorize 344 templates. We also need to find a bit flip in the URL of the Ubuntu or Debian update servers (depending on the target VM’s distribution) in the page cache entry for apt’s `sources.list` file. For `ubuntu.com`, 29 templates result in a valid domain name, and for `debian.org`, 26 templates result in a valid domain name. Table 5.2 shows examples of domains that are one bit flip away from `ubuntu.com`.

Performing the update attack on our Rowhammer testbed, we could trigger a bit flip in the page cache entry of sources `sources.list` in 212 seconds, converting `ubuntu.com` to `ubunvu.com`, a domain which we control. Further, we could trigger a bit flip in the page cache entry of `trusted.gpg` that changed one of the RSA public keys to one that we had pre-computed a factorization in 352 seconds. At this point, we manually sign the malicious package with our GPG private key that corresponds to the mutated public key. When the victim then updates the package database and upgrades the packages, the malicious package is downloaded and installed without warning. Since the current version of dFFS runs these steps sequentially, the entire end-to-end attack took 566 seconds. We have prepared a video of this attack which is available at: <https://vusec.net/projects/flip-feng-shui>

Growingly concerned about the impact of such practical attacks, we conservatively registered all the possible domains from our Ubuntu/Debian list.

5.4 RSA Modulus Factorization

Figure 7 shows the average probability of successful factorizations based on the amount of available compute

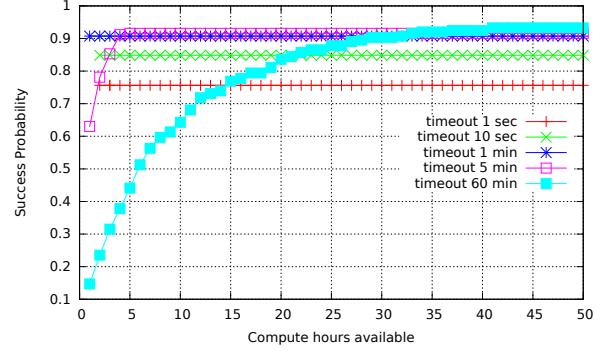


Figure 7: Compute power and factorization timeout tradeoff for 2048-bit RSA keys.

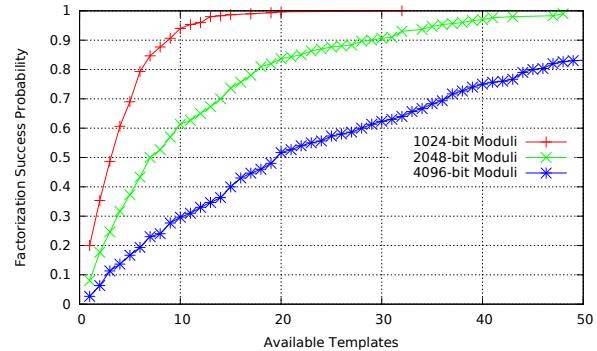


Figure 8: CDF of success rate with increasing templates.

hours. We generated this graph using 200 randomly generated 2048-bit RSA keys, each with a bit flip in 50 distinct trials (i.e., 10,000 keys, each with a bit flip). For this experiment, we relied on the ECM factorization tool, discussed in Section 3, and varied its user-controlled timeout parameter between one second and one hour. For example, with a timeout of one second for a key with a bit flip, we either timeout or the factorization succeeds immediately. In both cases, we move on to the next trial of the same key with a different bit flip.

This graph shows that, with 50 bit flips, the average factorization success probability is between 0.76 for a timeout of one second and 0.93 for a timeout of one hour. Note that, for example, with a timeout of one second, we can try 50 templates in less than 50 seconds, while achieving a successful factorization in as many as 76% of the public keys. A timeout of one minute provides a reasonable tradeoff and can achieve a success rate of 91% for 2048-bit RSA keys.

Figure 8 shows the cumulative success probability of factorization as more templates become available for 1024-bit, 2048-bit and 4096-bit keys. For 4096-bit keys, we need around 50 templates to be able to factorize a key with high probability (0.85) with a 1-hour timeout.

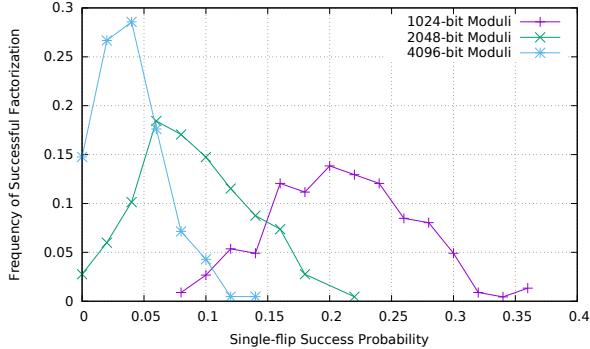


Figure 9: Probability mass function of successful factorizations with one flip.

With bit-flipped 2048-bit RSA public keys, with only 48 templates, we achieved a success probability of 0.99 with a 1 hour timeout. This proves that for 2048-bit keys (`ssh-keygen`'s default), *only a very small fraction of the templates from our testbed is necessary for a successful factorization*. For 1024-bit keys, we found a successful factorization for all keys after just 32 templates.

Some DRAM modules may only have a small number of bit flips [34], so an interesting question is: what is the chance of achieving a factorization using only a single template? Figure 9 answers this question for 1024-bit, 2048-bit and 4096-bit moduli separately. To interpret the figure, fix a point on the horizontal axis: this is the probability of a successful factorization using a single bit flip within 1 hour. Now read the corresponding value on the vertical axis, which shows the probability that a public key follows this success rate. For example, on average, 15% of 2048-bit RSA public keys can be factored using only a single bit flip with probability 0.1. As is expected, the probability to factor 4096-bit keys with the same 1-hour timeout is lower, and for 1024-bit keys higher. The fact that the distributions are centered around roughly 0.22, 0.11, and 0.055 are consistent with our analytical results in 3, which predict the factorization cost is linear in the bitlength of the modulus.

6 Mitigations

Mitigating Flip Feng Shui is not straightforward as hardware reliability bugs become prevalent. While there is obviously need for new testing methods and certification on the hardware manufacturer's side [4], software needs to adapt to fit Flip Feng Shui in its threat model. In this section, we first discuss concrete mitigations against dFFS before suggesting how to improve software to counter FFS attacks.

Table 2: Memory savings with different dedup strategies.

Strategy	Required memory	Savings
No dedup	506 GB	0%
Zero-page dedup	271 GB	46%
Full dedup	108 GB	79%

6.1 Defending against dFFS

We discuss both hardware and software solutions for defending against dFFS.

6.1.1 In Hardware

We recommend DRAM consumers perform extensive Rowhammer testing [2] to identify vulnerable DRAM modules. These DRAM modules should be replaced, but if this is not possible, reducing DRAM refresh intervals (e.g., by half) may be sufficient to protect against Rowhammer [51]. However, this also reduces DRAM performance and consumes additional power.

Another option is to rely on memory with error-correcting codes (ECC) to protect against single bit flips. Unfortunately, we have observed that Rowhammer can occasionally induce multiple flips in a single 64-bit word confirming the findings of the original Rowhammer paper [34]. These multi-flips can cause corruption even in presence of ECC. More expensive multi-ECC DIMMs can protect against multiple bit flips, but it is still unclear whether they can completely mitigate Rowhammer.

A more promising technology is *directed row refresh*, which is implemented in low-power DDR4 [7] (LPDDR4) and some DDR4 implementations. LPDDR4 counts the number of activations of each row and, when this number grows beyond a particular threshold, it refreshes the adjunct rows, preventing cell charges from falling below the error margin. Newer Intel processors support a similar feature for DDR3, but require compliant DIMMs. While these fixes mitigate Rowhammer, replacing most of current DDR3 deployments with LPDDR4 or secure DDR4 DIMMs (some DDR4 DIMMs are reported to be vulnerable to Rowhammer [1]), is not economically feasible as it requires compatible mainboards and processors. As a result, a software solution is necessary for mitigating Rowhammer in current deployments.

6.1.2 In Software

The most obvious mitigation against dFFS is disabling memory deduplication. In fact, this is what we recommend in security-sensitive environments. Disabling memory deduplication completely, however, wastes a

substantial amount of physical memory that can be saved otherwise [6, 46, 54].

Previous work [12] showed that deduplicating zero pages alone can retain between 84% and 93% of the benefits of full deduplication in a browser setting. Limiting deduplication to zero pages and isolating their Rowhammer-prone surrounding rows was our first mitigation attempt. To understand whether zero-page deduplication retains sufficient memory saving benefits in a cloud setting, we performed a large-scale memory deduplication study using 1,011 memory snapshots of *different* VMs from community VM images of Windows Azure [48]. Table 2 presents our results. Unfortunately, zero-page deduplication only saves 46% of the potential 79%. This suggests that deduplicating zero pages alone is insufficient to eradicate the wasteful redundancy in current cloud deployments. Hence, we need a better strategy that can retain the benefits of full memory deduplication without resulting in a memory massaging primitive for the attackers.

A strawman design A possible solution is to rely on a deduplication design that, for every merge operation, randomly allocates a new physical page to back the existing duplicate pages. When merge operations with existing shared pages occur, such design would need to randomly select a new physical page and update all the page-table mappings for all the sharing parties.

This strawman design eliminates the memory massaging primitive that is necessary for dFFS under normal circumstances. However, this may be insufficient if an attacker can find different primitives to control the physical memory layout. For example, the attacker’s VM can corner the kernel’s page allocator into allocating pages with predictable patterns if it can force the host kernel into an out-of-memory (OOM) situation. This is not difficult if the host relies on over-committed memory to pack more VMs than available RAM, a practice which is common in cloud settings and naturally enabled by memory deduplication. For example, the attacker can trigger a massive number of unmerge operations and cause the host kernel to approach an OOM situation. At this point, the attacker can release vulnerable memory pages to the allocator, craft a page with the same contents as the victim page, and wait for a merge operation. Due to the near-OOM situation, the merge operation happens almost instantly, forcing the host kernel to predictably pick one of the previously released vulnerable memory pages (i.e., templates) to back the existing duplicate pages (the crafted page and the victim page). At this stage, the attacker has again, in effect, a memory massaging primitive.

A better design To improve on the strawman design, the host needs to ensure enough memory is available not

to get cornered into predictable physical memory reuse patterns. Given a desired level of entropy h , and the number of merged pages m_i for the i th VM, the host needs to ensure $A = 2^h + \text{Max}(m_i)$ memory pages are available or can easily become available (e.g., page cache) to the kernel’s page allocator at all times. With an adequate choice of h , it may become difficult for an attacker to control the behavior of the memory deduplication system. We have left the study of the right parameters for h and the projected A for real systems to future work. We also note that balancing entropy, memory, and performance when supporting a truly random and deduplication-enabled physical memory allocator is challenging, and a promising direction for future work.

6.2 Mitigating FFS at the Software Layer

The attacks presented in this paper provide worrisome evidence that even the most security-sensitive software packages used in production account for no attacker-controlled bit flips as part of their threat model. While there is certainly room for further research in this direction, based on our experience, we formulate a number of suggestions to improve current practices:

- Security-sensitive information needs to be checked for integrity in software right before use to ensure the window of corruption is small. In all the cases we analyzed, such integrity checks would be placed on a slow path with essentially no application performance impact.
- Certificate chain formats such as X.509 are automatically integrity checked as certificates are always signed [17]. This is a significant side benefit of a certification chain with self-signatures.
- The file system, due to the presence of the page cache, should not be trusted. Sensitive information on stable storage should include integrity or authenticity information (i.e., a security signature) for verification purposes. In fact, this simple defense mechanism would stop the two dFFS attacks that we presented in this paper.
- Low-level operating system optimizations should be included with extra care. Much recent work [11, 12, 29, 40, 58] shows that benign kernel optimizations such as transparent huge pages, vsyscall pages, and memory deduplication can become dangerous tools in the hands of a capable attacker. In the case of FFS, any feature that allows an untrusted entity to control the layout or reuse of data in physical memory may provide an attacker with a memory massaging primitive to mount our attacks.

7 Related Work

We categorize related work into three distinct groups discussed below.

7.1 Rowhammer Exploitation

Pioneering work on the Rowhammer bug already warned about its potential security implications [34]. One year later, Seaborn published the first two concrete Rowhammer exploits, in the form of escaping the Google Native Client (NaCl) sandbox and escalating local privileges on Linux [51]. Interestingly, Seaborn’s privilege escalation exploit relies on a weak form of memory massaging by probabilistically forcing a OOMing kernel to reuse physical pages released from user space. dFFS, in contrast, relies on a deterministic memory massaging primitive to map pages from co-hosted VMs and mount fully reliable attacks. In addition, while mapping pages from kernel space for local privilege escalation is possible, dFFS enables a much broader range of attacks over nearly arbitrary physical memory.

Furthermore, Seaborn’s exploits relied on Intel x86’s CLFLUSH instruction to evict a cache line from the CPU caches in order to read directly from DRAM. For mitigation purposes, CLFLUSH was disabled in NaCl and the same solution was suggested for native CPUs via a microcode update. In response to the local privilege exploit, Linux disabled unprivileged access to virtual-to-physical memory mapping information (i.e., `/proc/self/pagemap`) used in the exploit to perform double-sided Rowhammer. Gruss et al. [30], however, showed that it is possible to perform double-sided Rowhammer from the browser, without CLFLUSH, and without pagemap, using cache eviction sets and transparent huge pages (THP). dFFS relies on nested THP (both in the host and in the guest) for reliable double-sided Rowhammer. In our previous work [12], we took the next step and implemented the first reliable Rowhammer exploit in the Microsoft Edge browser. Our exploit induces a bit flip in the control structure of a JavaScript object for pivoting to an attacker-controlled counterfeit object. The counterfeit object provides the attackers with arbitrary memory read and write primitives inside the browser.

All the attacks mentioned above rely on one key assumption: the attacker already *owns* the physical memory of the victim to make Rowhammer exploitation possible. In this paper, we demonstrated that, by abusing modern memory management features, it is possible to completely lift this assumption with alarming consequences. Using FFS, an attacker can seize control of nearly arbitrary physical memory in the software stack, for example compromising co-hosted VMs in complete absence of software vulnerabilities.

7.2 Memory Massaging

Sotirov [55] demonstrates the power of controlling virtual memory allocations in JavaScript, bypassing many protections against memory errors with a technique called Heap Feng Shui. Mandt [41] demonstrates that it is possible to control reuse patterns in the Windows 7 kernel heap allocator in order to bypass the default memory protections against heap-based attacks in the kernel. Inspired by these techniques, our Flip Feng Shui demonstrates that an attacker abusing benign and widespread memory management mechanisms allows a single bit flip to become a surprisingly dangerous attack primitive over physical memory.

Memory spraying techniques [25, 33, 47, 50] allocate a large number of objects in order to make the layout of memory predictable for exploitation purposes, similar, in spirit, to FFS. Govindavajhala and Appel [28] sprayed the entire memory of a machine with specially-crafted Java objects and showed that 70% of the bit flips caused by rare events cosmic rays and such will allow them to escape the Java sandbox. This attack is by its nature probabilistic and, unlike FFS, does not allow for fully controllable exploitation.

Memory deduplication side channels have been previously abused to craft increasingly sophisticated information disclosure attacks [8, 12, 29, 32, 43, 56]. In this paper, we demonstrate that memory deduplication has even stronger security implications than previously shown. FFS can abuse memory deduplication to perform attacker-controlled page-table updates and craft a memory massaging primitive for reliable hardware bit flip exploitation.

7.3 Breaking Weakened Cryptosystems

Fault attacks have been introduced in cryptography by Boneh et al. [9]; their attack was highly effective against implementations of RSA that use the Chinese Remainder Theorem. Since then, many variants of fault attacks against cryptographic implementations have been described as well as countermeasures against these attacks. Seifert was the first to consider attacks in which faults were introduced in the RSA modulus [52]; his goal was limited to forging signatures. Brier et al. [14] have extended his work to sophisticated methods to recover the private key; they consider a setting of uncontrollable faults and require many hundreds to even tens of thousands of faults. In our attack setting, the attacker can choose the location and observe the modulus, which reduces the overhead substantially.

In the case of Diffie-Hellman, the risk of using it with moduli that are not strong primes or hard-to-factor integers was well understood and debated extensively dur-

ing the RSA versus DSA controversy in the early 1990s (e.g., in a panel at Eurocrypt’92 [18]). Van Oorschot and Wiener showed how a group order with small factors can interact badly with the use of small Diffie-Hellman exponents [57]. In 2015, the Logjam attack [3] raised new interest in the potential weaknesses of Diffie-Hellman parameters.

In this paper, we performed a formal cryptanalysis of RSA public keys in the presence of bit flips. Our evaluation of dFFS with bit-flipped default 2048-bit RSA public keys confirmed our theoretical results. dFFS can induce bit flips in RSA public keys and factorize 99% of the resulting 2048-bit keys given enough Rowhammer-induced bit flips. We further showed that we could factor 4.2% of the two 4096 bit **Ubuntu Archive Automatic Signing Keys** with a bit flip. This allowed us to generate enough templates to successfully trick a victim VM into installing our packages. For completeness, we also included a formal cryptanalysis of Diffie-Hellman exponents in the presence of bit flips in Appendix A.

8 Conclusions

Hardware bit flips are commonly perceived as a vehicle of production software failures with limited exploitation power in practice. In this paper, we challenged common belief and demonstrated that an attacker armed with Flip Feng Shui (FFS) primitives can mount devastatingly powerful end-to-end attacks even in complete absence of software vulnerabilities. Our FFS implementation (dFFS) combines hardware bit flips with novel memory templating and massaging primitives, allowing an attacker to controllably seize control of arbitrary physical memory with very few practical constraints.

We used dFFS to mount practical attacks against widely used cryptographic systems in production clouds. Our attacks allow an attacker to completely compromise co-hosted cloud VMs with relatively little effort. Even more worryingly, we believe Flip Feng Shui can be used in several more forms and applications pervasively in the software stack, urging the systems security community to devote immediate attention to this emerging threat.

Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to the relevant parties.

Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback. This work was sup-

ported by Netherlands Organisation for Scientific Research through the NWO 639.023.309 VICI “Dowsing” project, Research Council KU Leuven under project C16/15/058, the FWO grant G.0130.13N, and by the European Commission through projects H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 644571 and H2020 ICT-2014-645622 “PQCRIPTO”.

References

- [1] DDR4 Rowhammer mitigation. <http://www.passmark.com/forum/showthread.php?5301-Rowhammer-mitigation&p=19553>. Accessed on 17.2.2016.
- [2] Troubleshooting Memory Errors – MemTest86. <http://www.memtest86.com/troubleshooting.htm>. Accessed on 17.2.2016.
- [3] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. CCS’15, 2015.
- [4] Barbara P. Aichinger. DDR Compliance Testing - Its time has come! In *JEDEC’s Server Memory Forum*, 2014.
- [5] Andrea Arcangeli. Transparent hugepage support. *KVM Forum*, 2010.
- [6] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. OLS’09, 2009.
- [7] JEDEC Solid State Technology Association. Low Power Double Data 4 (LPDDR4). JESD209-4A, Nov 2015.
- [8] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently Breaking ASLR in the Cloud. WOOT’15, 2015.
- [9] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14(2), 2001.
- [10] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.

- [11] Erik Bosman and Herbert Bos. Framing signals—a return to portable shellcode. SP’14.
- [12] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. SP’16, 2016.
- [13] Cyril Bouvier, Pierrick Gaudry, Laurent Imbert, Hamza Jeljeli, and Emmanuel Thomé. Discrete logarithms in $GF(p)$ – 180 digits. <https://listserv.nodak.edu/cgi-bin/wa.exe?A2=ind1406&L=NMBRTHRY&F=&S=&P=3161>. June 2014.
- [14] Eric Brier, Benoît Chevallier-Mames, Mathieu Ciet, and Christophe Clavier. Why one should also secure RSA public key elements. CHES’06, 2006.
- [15] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow Bending: On the Effectiveness of Control-flow Integrity. SEC’15, 2015.
- [16] Cristian Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4), 2003.
- [17] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Technical report, May 2008.
- [18] Yvo Desmedt, Peter Landrock, Arjen K. Lenstra, Kevin S. McCurley, Andrew M. Odlyzko, Rainer A. Rueppel, and Miles E. Smid. The Eurocrypt ’92 Controversial Issue: Trapdoor Primes and Moduli (Panel). Eurocrypt’92, 1992.
- [19] The Sage Developers. Sage Mathematics Software (Version). <http://www.sagemath.org>. Accessed on 17.2.2016.
- [20] Karl Dickman. On the frequency of numbers containing prime factors of a certain relative magnitude. *Arkiv forr Matematik, Astronomi och Fysik*, 1930.
- [21] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), 1976.
- [22] Paul Erdős and Mark Kac. The Gaussian Law of Errors in the Theory of Additive Number Theoretic Functions. *American Journal of Mathematics*, 62(1), 1940.
- [23] Chris Evans. The poisoned NUL byte, 2014 edition. <http://googleprojectzero.blogspot.nl/2014/08/the-poisoned-nul-byte-2014-edition.html>. Accessed on 17.2.2016.
- [24] Justin N. Ferguson. Understanding the heap by breaking it. In *Black Hat USA*, 2007.
- [25] Francesco Gadaleta, Yves Younan, and Wouter Joosen. ESSoS’10, 2010.
- [26] Daniel Kahn Gillmor. pem2openpgp - translate PEM-encoded RSA keys to OpenPGP certificates. Accessed on 17.2.2016.
- [27] GitHub Developer – Public Keys. <https://developer.github.com/v3/users/keys/>. Accessed on 17.2.2016.
- [28] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. SP ’03, 2003.
- [29] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. ESORICS’15. 2015.
- [30] Daniel Gruss, Clementine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. DIMVA’16, 2016.
- [31] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security and Privacy Magazine, special issue of Cloud Security*, 8, 2010.
- [32] Gorka Irazoqui, Mehmet Sinan IncI, Thomas Eisenbarth, and Berk Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. PETS’15, 2015.
- [33] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. Ret2Dir: Rethinking Kernel Isolation. SEC’14, 2014.
- [34] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. ISCA’14, 2014.
- [35] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman

- J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. CRYPTO’10, 2010.
- [36] Donald E. Knuth and Luis Trabb-Pardo. Analysis of a Simple Factorization Algorithm. *Theoretical Computer Science*, 3(3), 1976.
- [37] Dmitrii Kuvaiskii, Rasha Faqeih, Pramod Bhatia, Pascal Felber, and Christof Fetzer. HAFT: Hardware-assisted Fault Tolerance. EuroSys’16, 2016.
- [38] Hendrik W. Lenstra. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126, 1987.
- [39] Dwayne Litzenberger. PyCrypto - The Python Cryptography Toolkit. <https://www.dlitz.net/software/pycrypto/>. Accessed on 17.2.2016.
- [40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. SP’15, 2015.
- [41] Tarjei Mandt. Kernel Pool Exploitation on Windows 7. In *Black Hat Europe*, 2011.
- [42] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. 1996.
- [43] R. Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. IPCCC’11, 2011.
- [44] Peter Pessl, Daniel Gruss, Clementine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. SEC’16, 2016.
- [45] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1), 1978.
- [46] Shashank Rachamalla, Dabadatta Mishra, and Purushottam Kulkarni. Share-o-meter: An empirical analysis of KSM based memory sharing in virtualized systems. HiPC’13, 2013.
- [47] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. SEC’09, 2009.
- [48] Kaveh Razavi, Gerrit van der Kolk, and Thilo Kielmann. Prebaked uVMs: Scalable, Instant VM Startup for IaaS Clouds. ICDCS ’15, 2015.
- [49] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2), 1978.
- [50] Jurgen Schmidt. JIT Spraying: Exploits to beat DEP and ASLR. In *Black Hat Europe*, 2010.
- [51] Mark Seaborn. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. In *Black Hat USA*, 2015.
- [52] Jean-Pierre Seifert. On authenticated computing and RSA-based authentication. CCS’05, 2005.
- [53] Noam Shalev, Eran Harpaz, Hagar Porat, Idit Keidar, and Yaron Weinsberg. CSR: Core Surprise Removal in Commodity Operating Systems. AS-PLOS’16, 2016.
- [54] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide Page Deduplication in Virtual Environments. HPDC’12, 2012.
- [55] Alexander Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe*, 2007.
- [56] Kuniyasu Suzuki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication As a Threat to the Guest OS. EUROSEC’11, 2011.
- [57] Paul C. van Oorschot and Michael J. Wiener. On Diffie-Hellman key agreement with short exponents. Eurocrypt’96, 1996.
- [58] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side-Channels for Untrusted Operating Systems. SP’15, 2015.

Appendix A Cryptanalysis of Diffie-Hellman with Bit Flips

This section describes how one can break Diffie-Hellman by flipping a bit in the modulus. Similar to RSA, Diffie-Hellman cryptosystem performs computations modulo n . In the Diffie-Hellman key agreement scheme [21], however, the modulus n is prime or $s = \gamma_1 = 1$. It is very common to choose strong primes, which means that $q = (n - 1)/2$ is also prime; this is also the approach taken by OpenSSH. Subsequently a generator g is chosen of order q . In the Diffie-Hellman protocol the client

chooses a random $x \in [1, n - 1]$ and computes $g^x \bmod n$ and the server chooses a random $y \in [1, n - 1]$ and computes $g^y \bmod n$. After exchanging these values, both parties can compute the shared secret $g^{xy} \bmod n$. The best known algorithm to recover the shared secret is to solve the discrete logarithm problem to find x or y using the GNFS, which has complexity $O(L_n[1/3, 1.92])$. For a 512-bit modulus n , the pre-computation cost is estimated to be about 10 core-years; individual discrete logarithms $\bmod n$ can subsequently be found in 10 minutes [3]. The current record is 596 bits [13]; again 1024 bits seems to be within reach of intelligence agencies [3].

By flipping a single bit of n , the parties compute $g^x \bmod n'$ and $g^y \bmod n'$. It is likely that recovering x or y is now much easier. If we flip the LSB, $n' = n - 1 = 2q$ with q prime and g will be a generator. In the other cases n' is a t -bit or $(t - 1)$ -bit odd integer; we conjecture that its factorization has the same form as that of a random odd integer of the same size. It is not necessarily the case the g is a generator $\bmod n'$, but with very high probability g has large multiplicative order.

The algorithm to compute a discrete logarithm in $Z_{n'}$ to recover x from $y = g^x \bmod n'$ requires two steps.

1. Step 1 is to compute the factorization of n' . This is the same problem as the one considered in Section 3.
2. Step 2 consists in computing the discrete logarithm of $g^x \bmod n'$: this can be done efficiently by computing the discrete logarithms modulo $g^x \bmod p_i'^{\tilde{\gamma}_i}$ and by combining the result using the Chinese remainder theorem. Note that except for the small primes, the $\tilde{\gamma}_i$ are expected to be equal to 1 with high probability. Discrete logarithms $\bmod p_i'^{\tilde{\gamma}_i}$ can in turn be computed starting from discrete logarithms $\bmod p_i'$ ($\tilde{\gamma}_i$ steps are required). If $p_i' - 1$ is smooth (that is, it is of the form $p_i' = \prod_{j=1}^r q_j^{\delta_j}$ with q_j small), the Pohlig-Hellman algorithm [45] can solve this problem in time $O\left(\sum_{j=1}^r \delta_j \sqrt{q_j}\right)$. If n' has prime factors p_i' for which $p_i' - 1$ is not smooth, we have to use for those primes GNFS with complexity $O(L_{p_i'}[1/3, 1.92])$.

The analysis is very similar to that of Section 3, with a difference that for RSA we can use ECM to find all small prime factors up to the second largest one p_2' . With a simple primality test we verify that the remaining integer is prime and if so the factorization is complete. However, in the case of the discrete logarithm algorithm we have to perform in Step 2 discrete logarithm computations modulo the largest prime p_1' . This means that if n' would prime (or a small multiple of a prime), Step 1 would be easy but we have not gained anything with the

bit flip operation. It is known that the expected bitlength of the largest prime factor p_1' of n' is $0.624 \cdot t$ [36] (0.624 is known as the Golomb–Dickman constant). A second number theoretic result by Dickman shows that the probability that all the prime factors p_i' of an integer n' are smaller than $n'^{1/u}$ has asymptotic probability u^{-u} [20].

For $t = 1024$, the expected size of the largest prime factor p_1' of n' is 639 bits and in turn the largest prime factor of $p_1' - 1$ is expected to be 399 bits ($1024 \cdot 0.624^2$). Note that $p_1' - 1$ can be factored efficiently using ECM as in the RSA case. If $p_1' - 1$ has 639 bits, the probability that it is smooth (say has factors less than 80 bits) is $8^{-8} = 2^{-24}$, hence Pohlig-Hellman cannot be applied. We have to revert to GNFS for a 399-bit integer. However, with probability $2^{-2} = 1/4$ all the factors of n' are smaller than 512 bits: in that case the largest prime factor of $p_1' - 1$ is expected to be 319 bits, but again with probability $1/4$ all prime factors are smaller than 256 bits. Hence with probability $1/16$ GNFS could solve the discrete logarithm in less than 1 core hour.

For $t = 2048$, the expected size of the largest prime factor p_1' of n' is 1278 bits and the largest prime factor of $p_1' - 1$ is expected to be 797 bits – this is well beyond the current GNFS record. However, with probability $3 \cdot 10^{-3} = 0.037$ all prime factors of n' are smaller than 638 bits. Factoring $p_1' - 1$ is feasible using ECM, given that its second largest prime factor is expected to be 134 bits. The largest prime factor of $p_1' - 1$ is expected to be 398 bits. The discrete logarithm problem modulo the largest factor can be solved using GNFS in about 1 core-month. With probability $4 \cdot 10^{-4} = 3.9 \cdot 10^{-3}$ all prime factors of n' are smaller than 512 bits, and in that case the largest prime factor of $p_1' - 1$ is expected to be 319 bits, which means that GNFS would require a few core-hours.

Even if it would not be feasible to compute the complete discrete logarithm there are special cases: if x or y have substantially fewer than t bits, it is sufficient to recover only some of the discrete logarithms $\bmod p_i'$ and the hardest discrete logarithm p_1 can perhaps be skipped; for more details, see [3, 57].

The main conclusion is that breaking discrete logarithms with the bit flip attack is more difficult than factorizing, but for 1024 bits an inexpensive attack is feasible, while for 2048 bits the attack would require a moderate computational effort, the results of which are widely applicable. It is worth noting that this analysis is applicable to the DH key agreement algorithm in use by OpenSSH, defaulting to 1536-bit DH group moduli in the current OpenSSH (7.2), bitflipped variants of which can be pre-computed by a moderately equipped attacker, and applied to all OpenSSH server installations. The consequences of such an attack are decryption of a session, including the password if used, adding another attractive facet to attacks already demonstrated in this paper.

One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation

Yuan Xiao

Xiaokuan Zhang

Yinqian Zhang

Radu Teodorescu

Department of Computer Science and Engineering

The Ohio State University

{*xiao.465, zhang.5840*}@*buckeyemail.osu.edu*, {*yinqian, teodores*}@*cse.ohio-state.edu*

Abstract

Row hammer attacks exploit electrical interactions between neighboring memory cells in high-density dynamic random-access memory (DRAM) to induce memory errors. By rapidly and repeatedly accessing DRAMs with specific patterns, an adversary with limited privilege on the target machine may trigger bit flips in memory regions that he has no permission to access directly. In this paper, we explore row hammer attacks in cross-VM settings, in which a malicious VM exploits bit flips induced by row hammer attacks to crack memory isolation enforced by virtualization. To do so with high fidelity, we develop novel techniques to determine the physical address mapping in DRAM modules at runtime (to improve the effectiveness of double-sided row hammer attacks), methods to exhaustively hammer a large fraction of physical memory from a guest VM (to collect exploitable vulnerable bits), and innovative approaches to break Xen paravirtualized memory isolation (to access arbitrary physical memory of the shared machine). Our study also suggests that the demonstrated row hammer attacks are applicable in modern public clouds where Xen paravirtualization technology is adopted. This shows that the presented cross-VM row hammer attacks are of practical importance.

1 Introduction

Security of software systems is built upon correctly implemented and executed hardware-software contracts. Violation of these contracts may lead to severe security breaches. For instance, operating system security relies on the assumption that data and code stored in the memory subsystems cannot be altered without mediation by the software running with system privileges (*e.g.*, OS kernels, hypervisors, *etc.*). However, the recently demonstrated row hammer attacks [23], which are capable of inducing hardware memory errors without access-

ing the target memory regions, invalidate this assumption, raising broad security concerns.

Row hammer attacks exploit a vulnerability in the design of dynamic random-access memory (DRAM). Modern high-capacity DRAM has very high memory cell density which leads to greater electrical interaction between neighboring cells. Electrical interference from neighboring cells can cause accelerated leakage of capacitor charges and, potentially, data loss. Although these so-called “disturbance errors” have been known for years, it has only recently been shown that these errors can be triggered by software. In particular, [23] has demonstrated that malicious programs may issue specially crafted memory access patterns, *e.g.*, repeated and rapid activation of the same DRAM rows, to increase the chances of causing a disturbance error in neighboring rows.

Row hammer vulnerabilities have been exploited in security attacks shortly after its discovery [4, 10, 16, 20]. In particular, Seaborn [4] demonstrated two privilege escalation attacks that exploit row hammer vulnerabilities: One escaped from Google’s NaCl sandbox and the other gained kernel memory accesses from userspace programs running on Linux operating systems. Other studies [10, 16, 20] aim to conduct row hammer attacks from high-level programming languages, *e.g.*, JavaScript, so that an adversary can induce memory errors and escalate privileges remotely, by injecting malicious JavaScript code into the target’s web traffic (*e.g.*, by hosting malicious websites, cross-site scripting, man-in-the-middle attacks, *etc.*).

In contrast to the client-side bit flip exploitations, server-side row hammer attacks are much less understood. One particularly interesting scenario where server-side row hammer attacks are of importance is in multi-tenant infrastructure clouds, where mutually-distrusting cloud tenants (*i.e.*, users of clouds) may co-locate their virtual machines (VM) on the same physical server, therefore sharing hardware resources, including

DRAMs. Although server-grade processors and more expensive DRAMs are believed to be less vulnerable to row hammer attacks [23], studies have suggested that even servers equipped with error correcting (ECC) memory are not immune to such attacks [12, 23].

In this paper, we aim to explore row hammer attacks in cross-VM settings, and shed some light on the security, or lack thereof, in multi-tenant infrastructure clouds. The goal of this research is *not* to extensively study how vulnerable the cloud servers are. Rather, we explore whether the isolation of cloud software systems—virtual machines and hypervisors—can be circumvented by row hammer attacks (and if so, how?), should the underlying hardware become vulnerable.

Towards this end, we demonstrate cross-VM row hammer attacks with high fidelity and determinism, which can be achieved in the following pipelined steps.

First, determine physical address mapping in DRAM. Double-sided row hammer attacks target a specific memory row by hammering its two neighboring rows to enhance the effectiveness of the attack [4, 23]. Conducting such attacks, however, requires knowledge of the physical memory mapping in DRAMs (*i.e.*, bits in physical addresses that determine memory channels, DIMMs, ranks, banks, and rows). This enables the identification of addresses in neighboring rows of the same bank. However such information is not publicly available for Intel processors and memory controllers. Moreover, the same memory controller may map physical addresses to DRAMs in different ways, depending on how DRAM modules are configured.

To address this issue, we developed a novel algorithm to determine the memory mapping at runtime (Section 3). Each bank in a DRAM chip has a row buffer that caches the most recently used row in a bank. Therefore, by alternately accessing two rows in the same bank, we expect a higher memory access latency due to row buffer conflicts. The increase in access latency serves as the basis for a *timing channel* which can be used to determine if two physical memory addresses are mapped to the same DRAM bank. Building on the timing-channel primitive, we developed a novel graph-based algorithm which models each bit in a physical address as a node in a graph and establishes relationships between nodes using memory access latency. We show that the algorithm is capable of accurately detecting the row bits, column bits and bank bits. We empirically show the algorithm can accurately identify the DRAM mapping schemes automatically within one or two minutes on the machines we tested.

Second, conduct effective double-sided row hammer attacks. With knowledge of the DRAM address mapping, we conduct double-sided row hammer attacks from

Xen guest VMs. We first empirically study which row hammer attack methods (*i.e.*, accessing memory with or without `mfence` instructions, see Section 4) are most effective and lead to most bit flips. Then, in order to guarantee that sufficient exploitable bit flips (*i.e.*, located at specific memory locations and can be repeatedly induced in row hammer attacks) are found, we conduct exhaustive row hammer attacks from a guest VM to test all DRAM rows that are accessible to the VM. Because each VM is limited to a small portion of the entire physical memory, we also develop methods to explore more physical memory than assigned to our VM initially. In addition, we design a safe mode that makes bit flips induced by row hammer attacks less likely to crash the system.

Third, crack memory isolation enforced by virtualization. Unlike prior work, which sprays large numbers of page tables and conducts random row hammer attacks hoping that bit flips will occur in a page table entry (PTE) [4], in our approach (Section 5), we use hypercalls to map page directories in the OS kernel of our own VM to physical pages containing memory cells that are vulnerable to row hammer attacks. We then conduct row hammer attacks to deterministically flip the vulnerable bit at anticipated positions in a page directory entry (PDE), making it point to a different page table. In the context of this paper, we call such attack techniques *page table replacement* attacks to indicate that the original page table has been replaced with a forged one. We empirically demonstrate in Section 6 that such attacks allow a Xen guest VM to have both read and write access to any memory pages on the machine. We demonstrate two examples to illustrate the power of the cross-VM row hammer attacks: private key exfiltration from an HTTPS web server and code injection to bypass password authentication of an OpenSSH server. We emphasize that with the attack techniques we propose in this paper, the attacker’s capability is only limited by imagination.

We note our attacks primarily target Xen paravirtualized VMs, which, although are gradually superseded by hardware-assisted virtualization, are still widely used as cloud substrates in public cloud like Amazon EC2. This offers the adversary easy-to-break targets on servers with vulnerable hardware. Given the existing evidence of successful co-location attacks in public clouds [30, 32], we recommend discontinuing the use of such virtualization technology in cloud hosting services.

Contributions. This paper makes the following contributions to the field:

- A novel graph-based algorithm incorporating timing-based analysis to automatically reverse engineer the mapping of the physical addresses in DRAMs.
- A novel *page table replacement* technique that allows a malicious guest VM to have read and write accesses

to arbitrary physical pages on the shared machine.

- Implementation of effective double-sided row hammer attacks from guest VMs, and a systematic evaluation of the proposed techniques.
- Demonstration of two concrete examples to illustrate the power of the cross-VM attacks: private key extraction from HTTPS servers and code injection into OpenSSH servers to bypass authentication.

Roadmap. We will first summarize related work in the field and introduce background knowledge to set the stage for our discussion (Section 2). We will then describe a novel graph-based algorithm for detecting physical address mapping in DRAMs (Section 3). We then present a few technical details in our row hammer attack implementation (Section 4) and a *page table replacement* attack that enables arbitrary cross-VM memory accesses (Section 5). Next, we evaluate the proposed techniques (Section 6). Finally, we discuss existing countermeasures (Section 7) and conclude (Section 8).

2 Background and Related Work

2.1 DRAM Architecture

Modern memory systems are generally organized in multiple memory channels, each handled by its own dedicated memory controller. A channel is partitioned into multiple ranks. A rank consists of several DRAM chips that work together to handle misses or refill requests from the processor’s last-level cache. Each rank is also partitioned into multiple banks. Each bank has a row buffer to store the last accessed row in that bank. All banks and ranks can generally support independent transactions, allowing parallel accesses to the DRAM chips. A typical memory system is illustrated in Figure 1.

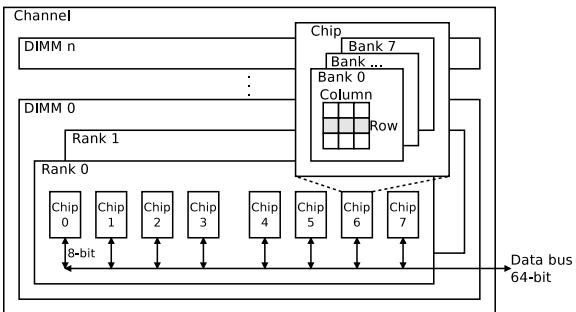


Figure 1: DRAM architecture.

DRAM chips are large arrays of memory cells with additional support logic for data access (read/write) and refresh circuitry used to maintain data integrity. Memory arrays are organized in rows (wordlines) and columns (bitlines) of memory cells.

Each memory cell consists of a capacitor that can be charged and discharged to store a 0 or a 1. An access transistor in each cell allows reads and writes to its content. The transistor is controlled through the wordline. When the wordline is activated, the content of all the capacitors on that row are discharged to the bitlines. Sense amplifier circuitry on each bitline amplifies the signal and stores the result in the row buffer.

Additional circuitry in the memory arrays includes address decoding logic to select rows and columns and internal counters to keep track of refresh cycles. In addition to the cells dedicated for data storage, DRAM chips often include additional storage for ECC (error-correction codes) or parity bits, to enable detection and/or correction of errors in the data array.

DRAM Refresh. The charge in the DRAM cell capacitor drains over time due to leakage current. To prevent data loss the content of the cell requires periodic “refresh.” The refresh interval ranges between 32 and 64 milliseconds and is specified as part of the DDR memory standard. Refresh operations are issued at rank granularity in recent designs. Before issuing a refresh operation, the memory controller precharges all banks in the rank. It then issues a single refresh command to the rank. DRAM chips maintain a row counter to keep track of the last row that was refreshed – this row counter is used to determine the rows that must be refreshed next.

DRAM address mapping. Given a physical memory address, the location of the data in the DRAM chips is determined by the DRAM address mapping schemes used by the memory controllers. This information, while available for some processors [3], is not revealed by major chip companies like Intel or ARM. Some preliminary exploration to determine DRAM address mapping on older Intel processors has been conducted by Seaborn [5]. Concurrently to our work, Pessl et al. [29] proposed methods to reverse-engineer physical address mapping in DRAM on both Intel and ARM platforms. Similar to our work, a timing-based approach was used to determine whether two addresses were mapped to two different rows of the same DRAM bank. Unlike our work, brute-force approaches were taken to (1) collect sets of memory addresses that are mapped to the same banks by randomly selecting addresses from a large memory pool and conducting the timing-based tests to cluster them, and (2) to determine the XOR-schemes (see Section 3) that are used by memory controllers, by testing all possible combinations of XOR-schemes against all sets of addresses.

The advantage of their approach over ours is that it exhaustively searches XOR-schemes without the need to reason about the complex logic behind them, as is done in our paper. However, our method targets specific bit

combinations and therefore is more efficient. Specially, it has been reported in [29] that it took about 20 minutes to reverse engineer the DRAM mapping on a normally-loaded system. Our approach, on the other hand, takes less than two minutes (see Section 6). In addition, Pessl et al. [29] also indicated that completeness is not guaranteed as it depends on random addresses. Hence, a complete test using their approach may take even longer.

2.2 Row Hammer and DRAM Bit Flips

Modern DRAM chips tend to have larger capacity, and hence higher density of memory cells. As a result, a memory cell may suffer from disturbance errors due to electrical interference from its neighboring cells. Moreover, certain memory access patterns, such as repeated and frequent row activation (“row hammering”), may easily trigger disturbance errors. The “row hammer” problem caught Intel’s attention as early as 2012 and was publicly disclosed around 2014 [13–15, 19]. Independent of Intel’s effort, Kim et al. [23] also reported that random bit flips can be observed by specially crafted memory access patterns induced by software programs.

The first practical row hammer exploit was published by Seaborn from Google [4], who demonstrated privilege escalation attacks exploiting row hammer vulnerabilities to break the sandbox of Google’s NaCl, and to obtain kernel memory accesses from userspace programs running on Linux operating systems. The study was quickly followed up by others [10, 16, 20], who demonstrated row hammer attacks using Javascript code, which meant that the attacks could be conducted without special privileges to execute binary code on target machines. This paper follows the same line of research, but our focus is server-side row hammer attacks, although some of the proposed techniques will also be useful in other contexts.

It has been claimed that server-grade processors and DRAM modules are less vulnerable to row hammer attacks [23], especially when the server is equipped with ECC-enabled DRAM modules. However, ECC is not the ultimate solution to such attacks. The most commonly used ECC memory modules implement single error-correction, double error-detection mechanisms, which can correct only one single-bit of errors within a 64-bit memory block, and detect (but not correct) 2-bit errors in the same 64-bit block. More bit errors cannot be detected and data and code in memory will be corrupted silently [23].

Dedicated defenses against row hammer vulnerabilities by new hardware designs have been studied in [22]. Particularly, Kim et al. [22] proposes Counter-Based Row Activation (CRA) and Probabilistic Row Activation (PRA) to address row hammer vulnerabilities. CRA counts the frequency of row activations and proactively

activates neighboring rows to refresh data; PRA enables memory controllers to activate neighboring rows with a small probability for every memory access.

3 DRAM Addressing

Prior work [4] has indicated that double-sided row hammer attacks are much more effective than single-sided ones. We therefore focus on developing a software tool to conduct double-sided row hammer attacks from within virtual machines. To make the attack possible, we first must find the physical memory address mapping in the target DRAMs, and do so without physical accesses to the machines. More precisely, we hope to determine which bits in a physical address specify its mapping to DRAM banks, rows and columns.

This information, however, is not available in the system configuration or in the memory controller or DRAM datasheets. Intel never discloses the mapping algorithm in their memory controllers; moreover, the same memory controller will likely map the same physical address to a different DRAM location if the number or size of DRAM chips is changed. Therefore, in this section, we present a method to reverse engineer the physical address mapping in DRAM at runtime. We call this procedure *bit detection*. It is important to note that we do not need to differentiate address bits for banks, ranks, or channels as long as their combination uniquely addresses the same DRAM bank.

3.1 A Timing-Channel Primitive

We resort to a known timing channel [27] to develop our bit detection primitive. The timing channel is established due to the row buffer in each DRAM bank. When two memory addresses mapped to the same DRAM bank in different rows are alternatively accessed in rapid succession, the accesses will be delayed due to conflicts in the row buffer (and subsequent eviction and reload of the row buffer). Therefore, by conducting fast and repeated accesses to two memory addresses, one can learn that the two address are located in different rows of the same bank if one observes longer access latency.

The algorithm is described in Algorithm 1. The input to the algorithm, LATENCY(), is a set of bit positions in the physical address space. We use I to denote the input. For example, $I = \{b_3, b_{17}\}$ represents the 3rd and 17th right-most bits of the physical address. LATENCY() randomly selects 100 pairs¹ of memory addresses from a large memory buffer, so that each pair of addresses differs only in the bit positions that are specified by the input, I : in each pair, one address has ‘1’s at all these bit

¹A sample size that is large enough to achieve statistical significance.

Algorithm 1: LATENCY()

```
Input: { $b_i$ } : a set of physical address bits
Output: Access latency: 1 (high) or 0 (low)
begin
    Randomly select 100 pairs of memory addresses that differ only in { $b_i$ }: One address in each pair with all  $b_i = 1$  and the other with all  $b_i = 0$ . Place all 100 pairs in address_pairs{ }
    for each pair k in address_pairs{ } do
        Start time measurement
        for j in  $10^3$  do
            Access both addresses in k
            cflush both addresses
            insert memory barrier
        end
        Stop time measurement
    end
    Return the average access latency compared to baselines
end
```

positions and the other address has ‘0’s at all these positions.

The algorithm enumerates each pair of addresses by measuring the average access latency to read each address once from memory. Specifically, it accesses both addresses and then issues `clflush` instructions to flush the cached copies out of the entire cache hierarchy. Hence the next memory access will reach the DRAM. A memory barrier is inserted right after the memory flush so that the next iteration will not start until the flush has been committed. The total access time is measured by issuing `rdtsc` instructions before and after the execution. The algorithm returns 1 (*high*) or 0 (*low*) to indicate the latency of memory accesses. $\text{LATENCY}()=1$ suggests the two physical addresses that differ only at the bit positions specified in the input are located on different rows of the same DRAM bank.

3.2 Graph-based Bit Detection Algorithms

Using the $\text{LATENCY}()$ timing-channel primitive we develop a set of graph-based bit detection algorithms. Specifically, we consider each bit in a physical address as a node in a graph; the edges in the graph are closely related to the results of $\text{LATENCY}()$: The set of bits are connected by edges, if, when used as the input to $\text{LATENCY}()$, yields high access latency. But the exact construction of these edges may differ in each of the graphs we build, as will be detailed shortly. We define all such nodes as set $V = \{b_i\}_{i \in [1, n]}$, where n is the total number of bits in a physical address on the target machine. In the following discussion, we use b_i to refer to an address bit position and a node interchangeably.

Our bit detection algorithms works under the assumption that Intel’s DRAM address mapping algorithms may use XOR-schemes to combine multiple bits in physical addresses to determine one of the bank bits. An XOR-scheme is a function which takes a set of bits as input

and outputs the XORed value of all the input bits. This assumption is true for Intel’s DRAM address mapping, which is evident according to prior studies [5, 25, 33]. Our empirical evaluation also confirms this assumption.

Detecting row bits and column bits. We first define a set of nodes $R = \{b_i | \text{LATENCY}(\{b_i\}) = 1, b_i \in V\}$. Because $\text{LATENCY}(\{b_i\}) = 1$, any two memory addresses that differ only in b_i are located in different rows of the same bank. Therefore, bit b_i determines in which rows the addresses are located, *i.e.*, b_i is a *row bit*. But as the two addresses are mapped to the same bank, b_i is not used to address DRAM banks.

Next, we define set $C = \{b_j | \text{LATENCY}(\{b_i, b_j\}) = 1, \forall b_i \in R, b_j \notin R\}$. It means that when accessing two addresses that differ only in a bit in C and a bit in R , we experience high latency in the $\text{LATENCY}()$ test—indicating that the two addresses are in the same bank but different rows. Therefore, the bits in C are not at all involved in DRAM bank indexing (otherwise changing bits in C will yield a memory address in a different bank). The bits in C are in fact *column bits* that determine which column in a row the address is mapped to.

Detecting bank bits in a single XOR-scheme. We consider an undirected graph G_1 constructed on the subset of nodes $V - R - C$. If $\text{LATENCY}(\{b_i, b_j\}) = 1$, node b_i is connected with node b_j by edge $e(b_i, b_j)$. There could be three types of connected components in such a graph: In the type I connected components, *only* two nodes are connected (Figure 2a). Because $\text{LATENCY}(\{b_i, b_j\}) = 1$, changing bits b_i and b_j together will yield an address in a different row of the same bank. Hence, at least one of b_i and b_j (usually only the more significant bit—the one on the left²) will be the row bit; the XOR of the two is a bank bit. More formally, if $e(b_i, b_j)$ is an edge in component type I (shown in Figure 2a), and $i > j$, b_i is a row bit, $b_i \oplus b_j$ determines one bank bit.

In the type II connected components, a set of nodes are connected through a hub node (Figure 2b). For instance, nodes b_j , b_k , and b_l are connected via node b_i . Particularly in Figure 2b, $i = 20, j = 15, k = 16, l = 17$. Due to the property of the $\text{LATENCY}()$ test, $b_i \oplus b_j$ must be a bank bit and at least one of the pair is a row bit. The same arguments apply to $b_i \oplus b_k$ and $b_i \oplus b_l$. We can safely deduce that $b_i \oplus b_j \oplus b_k \oplus b_l$ is a common XOR-scheme in which the four bits are involved: Otherwise, without loss of generality, we assume $b_i \oplus b_j \oplus b_k$ and $b_i \oplus b_l$ are two separate XOR-schemes. When two addresses differ only in b_i and b_j , although the value of $b_i \oplus b_j \oplus b_k$ does not change for the two addresses,

²The timing-channel approach cannot determine which bit is actually the row bit in this case. However, because memory controllers need to minimize row conflicts in the same bank, row bits are usually more significant bits in a physical address [5, 33]. Our hypothesis turned out to be valid in all the case studies we have conducted (see Table 1).

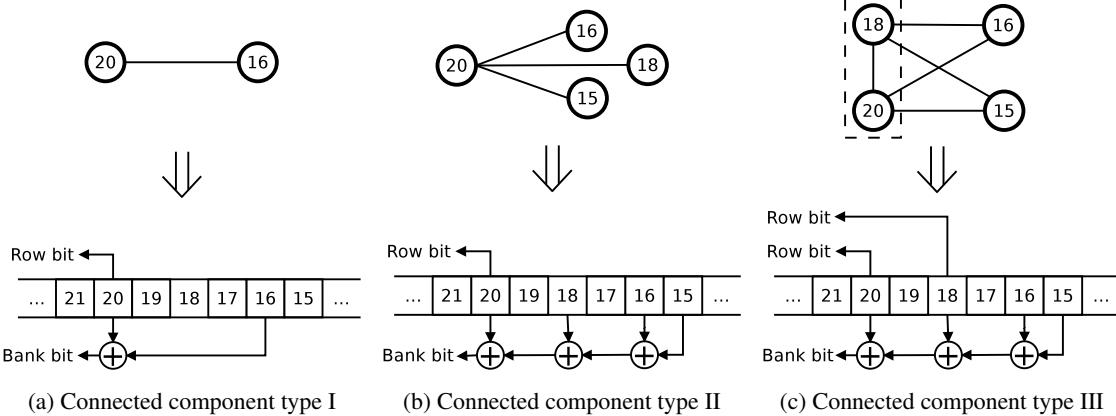


Figure 2: Detecting bits in a single XOR-scheme.

$b_i \oplus b_l$ will be different, thus making the two addresses in different banks. However, this conclusion contradicts the fact that $\text{LATENCY}(\{b_i, b_j\}) = 1$. Moreover, we can conclude that only b_i is the row bit, because otherwise if another bit is also a row bit, e.g., b_j , we should observe $\text{LATENCY}(\{b_j, b_k\}) = 1$ (because b_j and b_k are involved in the XOR-scheme $b_i \oplus b_j \oplus b_k \oplus b_l$ and b_j is a row bit). However that is not the case here. To summarize, if $e(b_i, b_j)$, $e(b_i, b_k)$ and $e(b_i, b_l)$ constitute a type II connected component in Figure 2b, b_i is a row bit and $b_i \oplus b_j \oplus b_k \oplus b_l$ determines one bank bit.

In the type III connected components, a clique of nodes replaces the single hub node in type II components—each node in the clique is connected to all other nodes in type III components (Figure 2c). As a simple example, we assume nodes b_i and b_j are connected by edge $e(b_i, b_j)$, and both of them are connected to nodes b_k and b_l , which are not connected directly. Particularly in Figure 2c, $i = 18$, $j = 20$, $k = 15$, $l = 16$. From the analysis of type II components, nodes b_i , b_k and b_l must follow that b_i is a row bit and $b_i \oplus b_k \oplus b_l$ determines one bank bit. Similarly, we can conclude that b_j is a row bit and $b_j \oplus b_k \oplus b_l$ determines one bank bit. Moreover, we can deduce that $b_i \oplus b_k \oplus b_l$ and $b_j \oplus b_k \oplus b_l$ determine the same bank bit, otherwise two addresses that differ in b_i and b_j will be in two different banks, which conflicts with $\text{LATENCY}(\{b_i, b_j\}) = 1$. Therefore, $b_i \oplus b_j \oplus b_k \oplus b_l$ is a bank bit. As such, in a type III component in Figure 2c, all nodes in the clique represent row bits, and the XOR-scheme that involves all bits in the components produces one bank bit.

Detecting bank bits in two XOR-schemes. On some processors, certain bits can be involved in more than one XOR-schemes. For instance, a bit b_i can be used in both $b_i \oplus b_j \oplus b_k$ and $b_i \oplus b_m \oplus b_n$. To detect such bit configuration, we consider another undirected graph G_2 constructed on the subset of nodes $V - R - C$. If

$\text{LATENCY}(\{b_i, b_j, b_m\}) = 1$, the three nodes are connected with each other by edges $e(b_i, b_j)$, $e(b_i, b_m)$, $e(b_j, b_m)$. If none of the three edges exist in graph G_1 —the graph we constructed in the single-XOR-scheme-bit detection—it means these three nodes are involved in two XOR-schemes $b_i \oplus b_j$ and $b_i \oplus b_m$: if two addresses differ in only two bits (out of the three), at least one of these two XOR-schemes will specify a different bank index; however, if two addresses differ in all three bits, the outcome of both XOR-schemes are the same for the two addresses, so they are in the same bank. One of these three bits (the most significant among the three) will be used in both XOR-schemes and serve as a row bit.

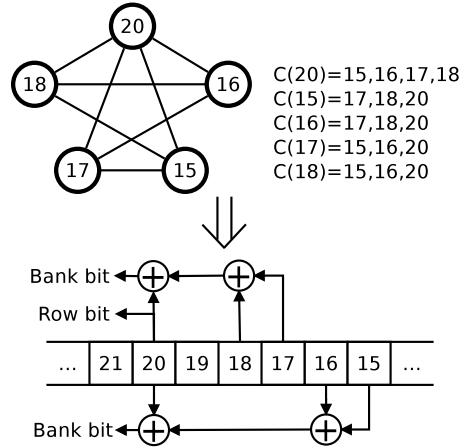


Figure 3: Detecting bits in two XOR-schemes.

Let's look at a more general example where five nodes are involved (Figure 3). In this example, the five nodes in the connected components of G_2 are b_{15} , b_{16} , b_{17} , b_{18} and b_{20} . They are connected by four triangles: (b_{15}, b_{18}, b_{20}) , (b_{16}, b_{17}, b_{20}) , (b_{16}, b_{18}, b_{20}) , (b_{15}, b_{17}, b_{20}) . Following the discussion in the previous paragraph, four XOR-schemes should be used

to index banks: $b_{20} \oplus b_{15}$, $b_{20} \oplus b_{16}$, $b_{20} \oplus b_{17}$ and $b_{20} \oplus b_{18}$. However, because $b_{20} \oplus b_{15}$ and $b_{20} \oplus b_{16}$ implies $\text{LATENCY}(\{b_{15}, b_{16}, b_{20}\}) = 1$, but a triangle (b_{15}, b_{16}, b_{20}) doesn't exist in our analysis, some of these XOR-schemes need to be merged together. To complete the analysis in the graph, we categorize nodes according to the set of nodes they are connected with. For instance, b_{20} is connected with $\{b_{15}, b_{16}, b_{17}, b_{18}\}$ (*i.e.*, $C(b_{20}) = b_{15}, b_{16}, b_{17}, b_{18}$). The node with the most connected neighbors is the one involved in both XOR-schemes (in this case, b_{20}) and therefore is a row bit. The nodes with the same set of neighboring nodes are used in the same XOR-scheme: b_{15} and b_{16} are both connected with $\{b_{17}, b_{18}, b_{20}\}$, and therefore one XOR-scheme will be $b_{15} \oplus b_{16} \oplus b_{20}$; similarly, the other XOR-scheme will be $b_{17} \oplus b_{18} \oplus b_{20}$.

Detecting bank bits in more XOR-schemes. If a bit is involved in more than two XOR-schemes, we can extend the method for detecting two XOR-schemes to detect it. Particularly, on the subset of nodes V – R – C, we enumerate all combination of four bits and look for $\text{LATENCY}(\{b_i, b_j, b_k, b_l\}) = 1$, which, following the reasoning steps in the prior paragraph, suggests that one of the bits is involved in three XOR-schemes. Again, we need to study the connected components to determine the configuration of actual XOR-schemes, which can be done by following a similar process as for two-XOR-scheme-bit detection. For concision we don't repeat the discussion here. However, it is worth noting we have not observed any bits that are used in more than two XOR-schemes on the machines we have tested.

4 Effective Row Hammer Attacks

In this section, we discuss several facets of constructing effective row hammer attacks in practice.

Row hammer code with or without `mfence`. prior work has proposed two ways of conducting row hammer attacks, pseudo code shown in Figure 4. Particularly, in each loop of the attacks, after accessing two memory blocks in two rows and flushing them out of the cache using `clflush` instructions, the attack code can choose to proceed with or without an `mfence` instruction before entering the next loop. The benefit of having an additional `mfence` instruction is to force the `clflush` instructions to take effect before the beginning of the next loop, while the downside is that it will slow down the execution of the program and thus reduce the frequency of memory accesses. We will empirically evaluate the two methods in Section 6.2.

Deterministic row hammer attacks. Prior studies [5] on row hammer exploitation randomly selected DRAM rows to attack and counted on luck to flip memory bits

```
loop:
    mov (X), %r10
    mov (Y), %r10
    clflush (X)
    clflush (Y)
    mfence
    jmp loop
```

(a) `clflush w/o mfence` (b) `clflush w/ mfence`

Figure 4: Pseudo code for row hammer attacks.

that happen to alter page tables. These approaches are non-deterministic and thus hard to guarantee success. In our paper, we propose to search exploitable bit flips that can be repeated in multiple runs. As will be discussed in Section 5, only bit flips at certain positions within a 64-bit memory block can be exploited; also, only a fraction of them are repeatable in row hammer attacks (we will empirically evaluate the fraction of vulnerable bits that are both exploitable and repeatable in Section 6.2.3). As such, on those less vulnerable machines, especially cloud servers, it is important to design methods to exhaustively search for vulnerabilities so that at least one of the vulnerable bit satisfies all the requirements.

Exhaustive row hammering. To enumerate as many DRAM rows as possible to look for vulnerable bits, we developed the following data structure and algorithm to conduct double-sided row hammer attacks on every row in every bank: Especially, as will be shown later in Table 1, some of the 12 least significant address bits are bank bits, which means the same 4KB memory page are not always mapped to the same row. As such, we designed a new data structure to represent memory blocks in the same row. Our key observation is that cache-line-aligned memory blocks are always kept in the same row for performance reasons. We call a cache-line-aligned, 64B in size, memory block a *memory unit*, which is the smallest unit of memory blocks for the purpose of book-keeping. We design a three dimension array: The first dimension represents the bank index, the second dimension is the row index and the third dimension stores an array of memory units mapped to the same row. For example, on a Sandy Bridge processor with 2 memory channels, 1 DIMM per channel, 1 rank per DIMM, and 8 banks per rank (totally 4GB memory), there are $2^4 = 16$ elements (*i.e.*, 2×8 banks) in the first dimension, $2^{16} = 65536$ elements (*i.e.*, number of rows per bank) in the second dimension, $2^7 = 128$ elements (*i.e.*, number of memory units per row) in the third dimension.

Another observation we had for conducting efficient row hammer attacks is to avoid hammering on rows in sequential order. According to our experiments, a recently-hammered row is harder to induce bit flips when its neighboring rows are hammered. This is probably be-

cause the cells in this row has been recently charged many times. Therefore, we targeted each row in a pseudorandom order. Specially, we first generate a pseudorandom permutation of all rows in a bank, and then sequentially test one row from each bank from the first to the last one and start over, where rows in the same bank are tested according to the pseudorandom order.

If no vulnerable bits were found in the first round of the attack, one can reboot the VM to obtain access to other DRAM rows and conduct row hammer attacks again. Even in public clouds, we found that rebooting the guest VMs will relaunch the VM on the same host, and possibly assigned to different (but largely overlapping) physical memory. As such, although each VM only has access to a small fraction of DRAM banks and rows, using such an approach will greatly increase the tested portion of the DRAM. We will empirically evaluate this technique in Section 6.2.

Safe mode. To safely conduct row hammer attacks without crashing the co-located VMs and the host machine, we optionally conduct the row hammer attacks in a safe mode: In Figure 5, only when we control all *memory units* in row n , $n+2$ and $n-2$ do we conduct the double-sided row hammer attacks on row $n+1$ and $n-1$. As rarely would the row hammer attacks affect rows beyond row $n \pm 2$, this method provides a safe mode to conducting row hammer attacks, which is particularly useful in attacks conducted in public clouds.

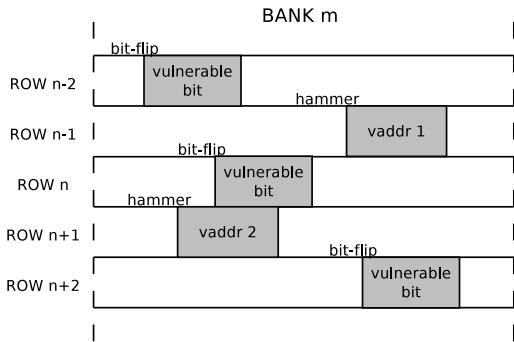


Figure 5: A safe mode of row hammer attacks.

5 Cracking Memory Isolation

In this section, we present methods to conduct cross-VM attacks enabled by DRAM row hammer vulnerabilities, which will allow a malicious paravirtualized VM to break VM isolation and compromise integrity and confidentiality of co-located VMs or even the VMM.

5.1 Xen Memory Management

Xen paravirtualization keeps three types of memory address spaces: a virtual address space for each process, a pseudo-physical address space for each VM, and a machine address space for the entire physical machine [17]. To be compatible with native OS kernels, a paravirtualized OS kernel (*e.g.*, already a part of mainstream Linux kernel) maintains a contiguous pseudo-physical memory address space; the mapping between pseudo-physical memory addresses and virtual addresses are maintained at page-granularity, following the same semantic as its non-virtualized counterparts. The major difference in a Xen paravirtualized VM is the page frame number (PFN) embedded in a page table entry (PTE): it is filled with machine addresses rather than pseudo-physical addresses. This is because Xen paravirtualization does not maintain a shadow page table in the hypervisor [17]. Address translation conducted by the CPU only traverses one layer of page tables. Such a memory management mechanism is called *direct paging* [11]. The mapping between each VM’s pseudo-physical memory pages to machine memory pages is also kept in the hypervisor, but guest VMs are allowed to query the mapping information by issuing hypercalls (*e.g.*, HYPERVISOR_memory_op()). The mapping between virtual memory pages, pseudo-physical memory pages and machine memory pages are illustrated in Figure 6.

To enable security isolation, the Xen hypervisor keeps track of the *type* of each memory page: page tables, segment descriptor page and writable pages. The hypervisor enforces an invariant that only *writable* pages can be modified by the guest VM. Whenever a page table hierarchy is loaded into the *CR3* register upon context switch, the hypervisor validates the memory types of the page tables to ensure the guest VM does not subvert the system by modifying the content of the page tables. On Intel’s x86-64 platforms, the page tables are organized in four-levels: PGD, PUD, PMD, PT³. Particularly of interest to us are the entries of PMD and PT, which are dubbed page directory entries (PDE) and page table entries (PTE), respectively. The structures of PDEs and PTEs are illustrated in Figure 7.

It is worthwhile noting that besides Xen paravirtualization technology, recent Xen hypervisors also support hardware-assisted virtualization, dubbed HVM in Xen’s term [18]. The memory management in Xen HVM is different from that in PVM in many aspects. Most notably, in HVM, guest VMs can no longer learn the physical address of the pseudo-physical memory pages, due to the intervention of a second-layer page table that is only ac-

³We use Linux terminology in this paper. Intel manuals call them page map level 4 (PML4, or PGD), page directory pointer tables (PDPT, or PUD), page directory tables (PDT, or PMD), page tables [6]. In Xen’s terminology, they are called L4, L3, L2 and L1 page tables [11].

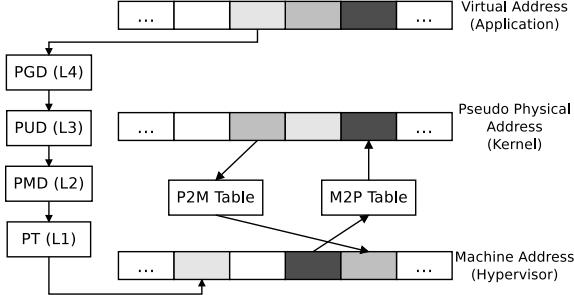


Figure 6: Memory management of Xen paravirtualized VMs.

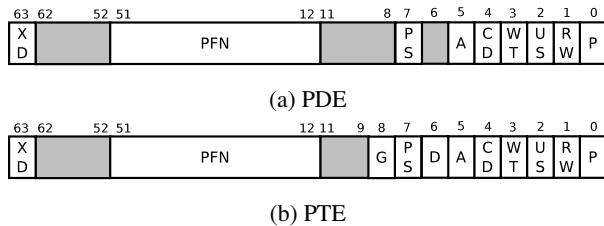


Figure 7: Structures of PDE, PTE.

cessible by the hypervisor. As such, much of the attack techniques discussed in this section only works in Xen paravirtualized machines.

5.2 Page Table Replacement Attacks

In this section, we present a method for a malicious guest VM to exploit the bit flips induced by row hammer attacks to gain arbitrary accesses to memory on the host machine. Instead of relying on an unreliable trial-and-error approach used in prior studies [4, 20], in which a large number of page tables are sprayed to increase the chances of bit flips taking place in PTEs, we propose a novel approach that, given a set of DRAM bit flips that an attacker could repeatedly induce, deterministically exploits the repeatable bit flips and gains access to physical memory pages of other VMs or even the hypervisor.

To access the entire machine address space with both read and write permissions, the attacker VM could do so by modifying a page table entry within its own VM so that the corresponding virtual address could be translated to a machine address belonging to other VMs or the hypervisor. However, direct modification of PTEs in this manner is prohibited. Every PTE update must go through the hypervisor via hypercalls, and thus will be declined. We propose a novel attack that achieves this goal by replacing the entire page tables in a guest VM without issuing hypercalls, which we call the *page table replacement* attacks.

For the convenience of discussion, we first define the

following primitives:

- $\text{Addr}(v)$ returns the machine address of a vulnerable bit.
- $\text{Offset}(v)$ returns the bitwise offset within a byte of a vulnerable bit (the right-most bit has an offset of 0).
- $\text{Direction}(v)$ could be one of $0 \rightarrow 1$, $1 \rightarrow 0$, or $0 \leftrightarrow 1$, indicating the most likely bit flip directions.
- $\text{Position}(v) = 64 - ((\text{Addr}(v) \% 8) \times 8 + 8 - \text{Offset}(v))$, indicating the index of the bit in a 64-bit aligned memory block (e.g., a page table entry). The right-most bit has a position of 0.
- $\text{Virt}(p)$ returns the virtual address of the beginning of a page p .
- $\text{Differ}(P_1, P_2)$ returns a set of indices of bits in which the machine addresses of two memory pages P_1 and P_2 differ.

Specially, when the vulnerable bit v satisfies $\text{Position}(v) \in [12, M]$, where M is the highest bit of the physical addresses on the target machine, the attacker could exploit the flippable bit to replace an existing page table with a carefully-crafted page table containing entries pointing to physical pages external to the guest VM via the following steps (Figure 8):

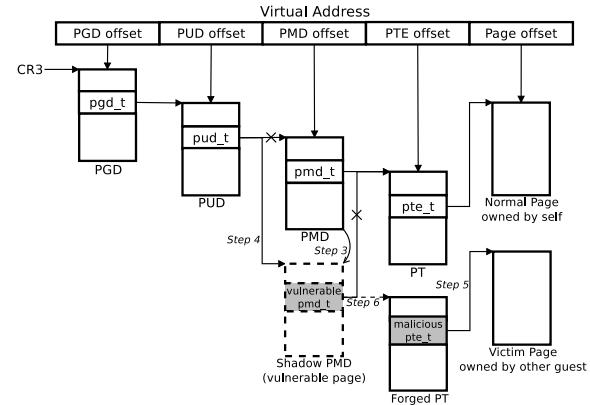


Figure 8: Page table replacement attacks.

- **Step 1:** In the attacker’s VM, allocate and map one virtual memory page (denoted p), so that the vulnerable bit v has the same page offset as one of the PFN bits in p ’s corresponding PDE. More accurately, $\text{Virt}(p)/2^{(9+12)} = \text{Addr}(v)/8 \bmod 2^9$. This can be achieved by allocating 1GB (*i.e.*, $512 \times 512 \times 4\text{KB}$) virtual pages in user space and map one of the pages that satisfies the requirement.
- **Step 2:** In guest kernel space, select two physical pages, P_1 and P_2 , where $\text{Differ}(P_1, P_2) = \{\text{Position}(v)\}$ and $\text{Position}(v)$ of P_1 is the original state of the vulnerable bit (*e.g.*, 0 if

- $\text{Direction}(v) = 0 \rightarrow 1$). Copy p 's PT to P_1 . Then deallocate all mappings to P_1 and make it read-only.
- **Step 3:** Copy p 's PMD to the physical page (denoted P_v) that contains the vulnerable bit v . Then change the PDE (on P_v) that contains v to point to P_1 . Then deallocate all mappings to P_v and make it read-only.
 - **Step 4:** Issue hypercalls to update p 's corresponding PUD entry with P_v 's machine address, so that P_v will become the new PMD. The Hypervisor will check the validity of the new PMD and all page tables it points to. Although p 's PDE has been changed to point to P_1 , because P_1 is exact the same as p 's original PT, this step will also pass the security check by the hypervisor.
 - **Step 5:** Construct fake PTEs on P_2 so that they point to physical pages outside the attacker VM. These are the target memory pages that the attacker would like to access.
 - **Step 6:** Conduct row hammer attacks on the two neighboring rows of the vulnerable bit v , until bit flip is observed. p 's PDE will be flipped so that it will point to P_2 instead of P_1 .
 - **Step 7:** Now the attacker can access p and the other 511 virtual pages controlled by the same page table P_2 to access physical memory outside his own VM. The attacker can also modify the PTEs in P_2 without issuing hypercalls as he has the write privilege on this forged page table.

Theoretically, $(52 - 12)/64 = 62.5\%$ vulnerable bits can be exploited in *page table replacement* attacks, regardless of flippable directions. In practice, because physical addresses on a machine is limited by the available physical memory, which is much less than the allowed $(2^{52} - 1)\text{B}$. For example, with 128GB memory, the most significant bit in a physical address is bit 38. Therefore the fraction of vulnerable bits that are exploitable is about 41%. We will empirically show the fraction of vulnerable bits that are exploitable in our attacks in Section 6.

6 Evaluation

In this section, we will first evaluate the effectiveness and efficiency of the bit detection algorithms (described in Section 3) in Section 6.1, our row hammer attacks (described in Section 4) in Section 6.2, and the cross-VM memory access attacks (described in Section 5) in Section 6.3.

6.1 Bit Detection Efficiency and Accuracy

We ran the bit detection algorithm detailed in Section 3 on a set of local machines. The processor and DRAM

configurations, together with the detected physical address mapping in the DRAMs, are shown in Table 1. For instance, on a machine equipped with an Intel Westmere processor, Xeon E5620, and one DRAM chip (with 2 memory channels, 1 DIMM, 2 ranks, 8 banks, and 2^{15} rows per bank), we ran our algorithm and found the bits that determine bank indices are $b_6 \oplus b_{16}$, b_{13} , b_{14} , b_{20} , b_{21} , and the bits that determine row indices are bits b_{16} to b_{19} , and bits b_{22} to b_{32} (totally 15 bits). We can see from these results that older processors, such as Westmere and Sandy Bridge, tend to have simpler XOR-schemes. More recent processors may have complex schemes (probably due to *channel hashing* [21]). For example, on an Intel Haswell Xeon E5-1607 v3 processor, we observed that complicated XOR-schemes, such as $b_7 \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{18} \oplus b_{26}$ and $b_8 \oplus b_{13} \oplus b_{15} \oplus b_{17} \oplus b_{27}$ are used to determine DRAM banks. Moreover, only on recent processors (e.g., Intel Broadwell Core i5-5300U) did we observe the same address bit involved in two XOR-schemes (e.g., b_{18} and b_{19}); other bits are at most used in one XOR-scheme. In addition, row bits are mostly contiguous bits, and on some processors can be split into two segments. For example, on an Intel Xeon E5-2640 v3 processor we tested on, the row bits are $b_{15} \sim b_{17}$ and $b_{21} \sim b_{35}$.

Efficiency evaluation. Figure 9 shows the execution time of the bit detection algorithms. Results for five local machines (Intel Sandy Bridge Core i3-2120 with 4GB memory, Intel Broadwell Core i5-5300U with 8GB memory, Intel Westmere Xeon E5620 with 4GB memory, Intel Haswell Xeon E5-2640 v3 with 32GB memory, and Intel Haswell Xeon E5-1607 v3 with 16GB memory) and three cloud machines (one machine in Cloudlab, Emulab d820, with 128GB memory, and two machines on Amazon EC2, one c1.medium instance and one c3.large instance, total memory size unknown) are shown in Figure 9. Most of these experiments can finish within one minute, with one exception of Xeon E5-2640 v3 which takes almost two minutes. The longer latency for testing E5-2640 v3 may be caused by its use of DDR4 memory, while the others are equipped with DDR3 memory chips.

Validation. Because Intel does not publish the memory mapping algorithms of their memory controllers, we do not have ground truth to validate our algorithm. However, we show that our algorithm is very likely to produce valid results for two reasons: First, in Table 1, the total number of bank bits and row bits detected are consistent with the DRAM configuration that we learned using several third-party software tools, including `dmidecode`, `decode-dimmms` and `HWiNFO64`. Second, we conducted double-sided row hammer attacks on some of the local machines we have in our lab: Machine A, Sandy Bridge

Processor Family	Processor Name	Channels	DIMMs	Ranks	Banks	Rows	Bank bits	Row bits
Westmere	Intel Xeon E5620	2	1	2	8	2^{15}	$b_6 \oplus b_{16}, b_{13}, b_{14}, b_{20}, b_{21}$	$b_{16} \sim b_{19}$ $b_{22} \sim b_{32}$
Sandy Bridge	Intel Core i3-2120	2	1	1	8	2^{15}	$b_6, b_{14} \oplus b_{17}, b_{15} \oplus b_{18}, b_{16} \oplus b_{19}$	$b_{17} \sim b_{31}$
Haswell	Intel Core i5-2500	2	1	1	8	2^{15}	$b_6, b_{14} \oplus b_{17}, b_{15} \oplus b_{18}, b_{16} \oplus b_{19}$	$b_{17} \sim b_{31}$
Haswell	Intel Xeon E5-1607 v3	4	1	1	8	2^{15}	$b_7 \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{18} \oplus b_{26},$ $b_8 \oplus b_{13} \oplus b_{15} \oplus b_{17} \oplus b_{27},$ $b_{19} \oplus b_{23}, b_{20} \oplus b_{24}, b_{21} \oplus b_{25}$	$b_{23} \sim b_{34}$
	Intel Xeon E5-2640 v3	2	1	2	16	2^{18}	$b_6 \oplus b_{21}, b_{13}, b_{34},$ $b_{18} \oplus b_{22}, b_{19} \oplus b_{23}, b_{20} \oplus b_{24}$	$b_{15} \sim b_{17}$ $b_{21} \sim b_{35}$
Broadwell	Intel Core i5-5300U	2	1	1	8	2^{16}	$b_7 \oplus b_8 \oplus b_9 \oplus b_{12} \oplus b_{13} \oplus b_{18} \oplus b_{19},$ $b_{14} \oplus b_{17}, b_{15} \oplus b_{18}, b_{16} \oplus b_{19}$	$b_{17} \sim b_{32}$

Table 1: Identifying physical address mapping in DRAMs.

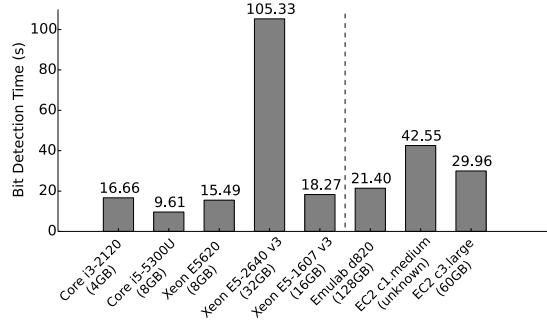


Figure 9: Efficiency of bit detection.

i3-2120, Machine B, Sandy Bridge i3-2120, Machine C, Sandy Bridge i5-2500, and Machine D, Broadwell i5-5300U⁴. Particularly on each of these machines, we indexed each row of the same bank from 1 to 2^k , where k is the number of detected row bits; the index of a row is given by the value presented by all row bits in the same order as they are in the physical address. Then we conducted row hammer attacks on row $n + 1$ and $n - 1$ of the same bank, where n ranged from 3 to $2^{15} - 2$. If the bit detection algorithm are correct, we should find more bit flips in row n than row $n + 2$ and $n - 2$, because double-sided row hammer attacks have been reported to be more effective [4]. It is apparent in Figure 10 that on all these machines, much more bit flips were found in row n than the other rows. For example, on machine A, 52.4% bit flips were found in row n , while only 28.6% and 19.0% flippable bits were found in row $n - 2$ and $n + 2$, respectively. These results suggest that our algorithm to detect the row bits and bank bits (including XOR-schemes) are consistent with the true configuration with the DRAM. We believe these evidence are strong enough to show the

validity of our bit detection method.

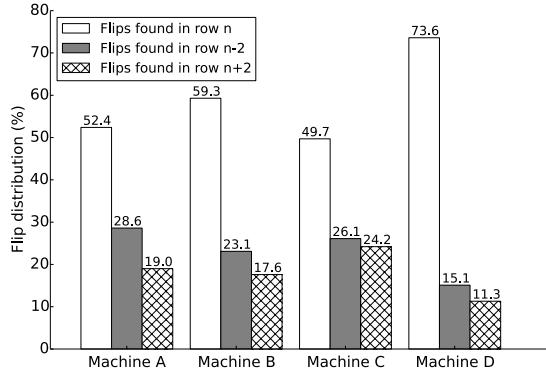


Figure 10: Location of bit flips in double-sided row hammer attacks. Row $n + 1$ and $n - 1$ are frequently accessed to induce disturbance errors.

6.2 Effectiveness of Row Hammer Attacks

We evaluated the effectiveness of our row hammer attacks in two aspects: (1) whether the attacker controlled physical memory can cover a significant portion of the overall physical memory on the machine, and (2) the number of bit flips induced by our double-sided row hammer attacks compared with single-sided attacks.

6.2.1 Physical Memory Coverage

We experimented on four servers to evaluate the physical memory coverage. The first machine is a desktop in our lab. It is equipped with a 3.3GHz Intel Core i3-2120 processor and 8GB of memory, of which 1GB is assigned to the virtual machine. The second machine is another desktop with a 3.7GHz Intel Core i5-2500 processor and 4GB of memory. The VM owns 1GB of the

⁴These set of machines, and the same naming convention, are also used in the following experiments.

memory. The third machine is a server in Cloudlab, which is equipped with a 2.2GHz Intel Xeon E5-4620 processor with 128GHz of memory. The VM runs on this machine is allowed to control 4GB of memory. The fourth machine is a dedicated cloud server in Amazon EC2. It has 128GB of memory and operates on a 2.8GHz Intel E5-2680 v2 processor. Our VM was allocated 8GB of memory.

We conducted the experiments as follows. On each of these VMs, we ran a program to measure the physical pages that are accessible to the guest VM. Then we rebooted our VM and measured the accessible physical memory again. After each reboot, some new physical pages will be observed (but some old pages will be deallocated from this VM). We rebooted the VM several times until no more new memory pages are observed after reboot. In Figure 11, the x-axis shows the number of VM reboots (the first launch counted as one reboot) and the y-axis shows the fraction of physical memory that can be accessed by the VM. In the two local machines, because no other VMs are competing for the physical memory, the sets of accessible pages are relatively stable. But still after reboots, more memory pages are accessible to the guest VMs. In the two cloud tests (one in EC2 and one in Cloudlab), the total physical memory sizes are very large (*i.e.*, 128GB). Although our VM were only allocated 6.25% (in the EC2 test) and 3.125% (in the Cloudlab test) physical memory initially, after several reboots, our VM could access as much as 17.8% (in the EC2 test) and 22.3% (in the Cloudlab test) of the total memory. The results suggest that row hammer attacks are possible to enumerate a large fraction of the physical memory even though the VM can only control a small portion of it at a time. Therefore, by doing so, the chances for a guest VM to induce exploitable and repeatable bit flips are not bound by the fixed size of physical memory allocated to the VM.

6.2.2 Row Hammer Induced Bit Flips

To show that our double-sided row hammer attacks are more effective than single-sided versions, we empirically test how fast each method can induce memory bit flips. In addition, we also tested with row hammer code both with and without `mfence` to empirically evaluate the effectiveness of the two types of attack techniques

Particularly, we implemented four types of row hammer attack tools: double-sided row hammer without `mfence` instruction, double-sided row hammer with `mfence`, single-sided row hammer without `mfence`, and single-sided row hammer with `mfence`. In Figure 12, we show the number of bit flips induced per hour by one of these approaches on four machines: Machine A, Sandy Bridge i3-2120, Machine B, Sandy Bridge

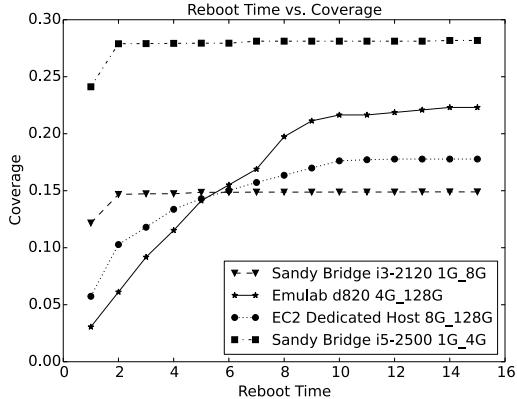


Figure 11: Physical memory coverage after VM rebooting.

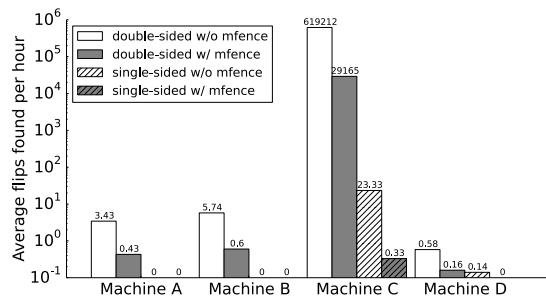


Figure 12: Efficiency of double-sided row hammer attacks.

i3-2120, Machine C, Sandy Bridge i5-2500, and Machine D, Broadwell i5-5300U (memory configurations are listed in Table 2).

We can see from the figure that our double-sided row hammer is much more effective than the single-sided row hammer attacks used in prior studies: Using single-sided attacks, on machine A and machine B, no bit flips could be observed, whether or not `mfence` was used. In contrast, using our double-sided row hammer attacks without `mfence`, 4 or 5 bits can be flipped per hour. On the most vulnerable machine C⁵, our double-sided row hammer attacks can find as many as over 600k bit flips per hour, while the best single-sided attacks can only find 23 bit flips per hour. We also find that row hammer without `mfence` is more effective than with it. The trend is apparent on all the four machines we tested on. As such, we conclude that although `mfence` ensures that all memory accesses reach the memory, the slowdown to the program execution it brings about reduces the effectiveness of row

⁵Some machines are expected to be more vulnerable than others (see Table 3, [23]), possibly due to higher memory density or lower DRAM refreshing frequency.

hammer attacks. Our double-sided row hammer attacks without `mfence` represent the most effective attack technique among the four.

While Figure 12 illustrates the rate of inducing bit flips, Table 2 demonstrates the overall effectiveness of our double-sided row hammer attacks (without `mfence`). Particularly, the total execution time of the experiments above and the total number of induced bit flips are shown in Table 2. In each of the tests we stopped the row hammer attacks once we have examined 50% of all DRAM rows (all rows that are accessible by the VM without reboot). We can see in the table the experiments took about 10 to 20 hours on machine A, B, and C. The total numbers of vulnerable bits found on machine A and B were 63 and 91, respectively. In contrast to zero bit flips induced by single-sided attacks that ran for 30 hours, our double-sided attacks make these machines vulnerable. On machine C, 5,622,445 vulnerable bits were found within 10 hours. Machine D is the least vulnerable among the four: only 25 vulnerable bits were found in about 43 hours. The results show that different machines are vulnerable to row hammer attacks to different extent.

Machine configuration	Execution time (hours)	Vulnerable bits found
(Machine A) Sandy Bridge i3-2120 (4GB)	18.37	63
(Machine B) Sandy Bridge i3-2120 (4GB)	15.85	91
(Machine C) Sandy Bridge i5-2500 (4GB)	9.08	5622445
(Machine D) Broadwell i5-5300U (8GB)	42.88	25

Table 2: Execution time and detected vulnerable bits in exhaustive row hammer attacks.

6.2.3 Vulnerable Bits Usability and Repeatability

We first report the fraction of vulnerable bits we found on the four machines, machine A, B, C and D (configurations listed in Table 2), that are usable in the *page table replacement* attacks we discussed in Section 5. The total number of bits that are used for analysis on these four machines are listed in Table 2⁶. The results are shown in Figure 13a: 36.5%, 31.9%, 32.8%, 40.0% of these bits are in the PFN range of a page table entry, thus are usable in *page table replacement* attacks.

Prior studies [23] have shown that many of the bit flips are repeatable. We try to confirm this claim in our own

⁶We selected a subset of vulnerable bits, 100031 vulnerable bits, on machine C for analysis because the entire set was too large to handle.

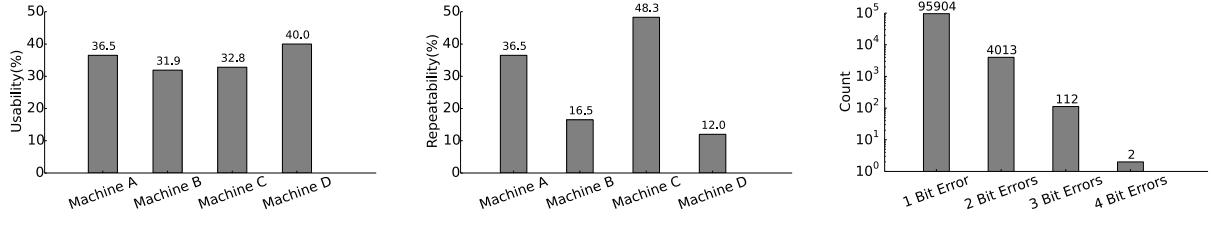
experiments. Specially, on these four machines, we repeated the row hammer attacks (10 times) against the rows in which vulnerable bits were found during the first sweep. We show, in Figure 13b, that 36.5%, 16.5%, 48.3%, and 12.0% of the vulnerable bits induced in the first run could be flipped again (at least once) on these four machines, respectively. These repeatable bit flips can be exploited in our cross-VM exploits.

In addition, on machine C, we have found more than one bit flippable within the same 64-bit memory block, which are beyond correction even with ECC memory. The distribution of vulnerable bits found in a 64-bit block is shown in Figure 13c. Particularly, we found 95904 single-bit errors, 4013 two-bit errors, 112 three-bit errors and 2 four-bit errors in the same 64-bit block.

6.3 Cross-VM Row Hammer Exploitation

We implemented our attack in a kernel module of Linux operating system (kernel version 3.13.0) that ran on Xen guest VMs. The hypervisor was Xen 4.5.1 (latest as of January 2016). We conducted the attacks on machine D, which is quipped with a Broadwell i5-5300U processor and 8GB of DRAM. However, we note that the attacks should also work on other machines and software versions as long as exploitable bits can be induced by row hammer attacks. Particularly, we demonstrated the power of the cross-VM row hammer attacks in two examples: In the first example, we demonstrated a confidentiality attack where the adversary exploited the techniques to steal TLS private keys from an Apache web server; in the second example, we showed an integrity attack, in which the attacker altered the program code of an OpenSSH server to bypass the user authentication and logged in the server without knowledge of credentials.

Arbitrary memory accesses. The first step of both attacks is to obtain arbitrary accesses to the target memory page. To do so, the adversary controlling a guest VM first runs the bit detection algorithm described in Section 3 to determine the row bits and bank bits of the machine, and then performs row hammer attacks until he finds a exploitable and repeatable bit flip at desired bit position—the PFN range of a PDE. We repeated the row hammer attacks 10 times and on average it took 2.13 hours to find the first useable bit flip. We emphasize machine D, the one we experimented with, is the least vulnerable machine among all (see Figure 12). Then the adversary replaces one of his own page tables with a forged one, using *page table replacement* attack techniques, and maps 512 of his virtual pages to 512 different physical pages. The adversary scans all these pages directly because they are mapped to his own address space. For each page, he compares the content of the page with a specific pattern. If the pattern is not found in these 512 pages, the ad-



(a) Vulnerable bits that are usable in *page table replacement* attacks. (b) Vulnerable bits that are repeatable after the first occurrence. (c) Distribution of vulnerable bits within the same 64-bit memory block.

Figure 13: Statistics of the induced flippable bits.

versary modifies the PTEs directly as he already has the write privilege on the forged page table, and searches in another 512 physical pages. The translation lookaside buffer (TLB) is flushed as needed to accommodate the page table changes.

To speed up the searching, the adversary obtained a list of machine page number (MFN) controlled by his own VM from `struct start_info.mfn_list` and excluded them from the list of physical pages to scan. As an extension of this implemented approach, the adversary may also reboot the VM several times to increase the physical memory space that is accessible to his own VM (as done in Section 4), thus reducing the search space of the victim. Alternatively, we also believe it is feasible to exploit cache-based side-channel analysis to learn the cache sets (physical address modulus the number of cache sets) of the targets [26] to narrow down the search space. We leave this optimization as future work.

6.3.1 Confidentiality Attacks

We show in this example that using the cross-VM row hammer attacks, the adversary may learn the private key of the Apache web servers of the neighboring VMs. Particularly, we set up two VMs on the same machine. The victim ran an Apache web server in which an HTTPS server was configured to support SSL/TLS using one pair of public/private keys. The attacker VM conducted the cross-VM row hammer attacks described above to obtain read access to the physical memory owned by the victim VM. When scanning each of the physical pages belonging to another VM, the adversary checked at each byte of the memory if it was the beginning of a `struct RSA`, by first checking if some of its member variables, such as `version` and `padding`, are integers, and others, such as `p`, `q`, `n` are pointers, and, if so, calling the `RSA_check_key()` function provided by OpenSSL. The function takes as argument a pointer to `struct RSA` and validates (1) whether `p` and `q` are both prime numbers, and (2) whether $n = p \times q$ and (3) whether $(x^e)^d \equiv x \pmod{n}$. If the location passes the checks, it

is the beginning of an RSA structure, the private key can be extracted. In fact, because at most memory locations, the basic checks will not pass, the expensive `RSA_check_key()` will not be called. If the adversary is lucky enough to successfully guess the machine address of the target memory page in the first trial, the average time to complete the attack was 0.32s (including the time to manipulate page tables, conduct row hammer attacks to induce the desired bit flip, read the memory page and check the validity of the private key, and write the extracted key to files). The overall execution time of the attack depends on the number of physical pages scanned before finding the target one, but on average scanning one additional memory pages took roughly 5ms.

6.3.2 Integrity Attacks

In this example, we show how to exploit row hammer vulnerabilities to log in an OpenSSH server without passwords. Particularly, the victim was the management domain in Xen, the Dom0. In our testbed, Dom0 is configured to use Pluggable Authentication Modules (PAM) for password authentication. PAM offers Linux operating systems a common authentication scheme that can be shared by different applications. Configuring `sshd` to use PAM is a common practice in Red Hat Linux [8]. We pre-configured one legitimate user on the OpenSSH server, and enabled both public key authentication and password authentication. The adversary controls a regular guest VM, a DomU, that ran on the machine. We assume the adversary has knowledge of the username and public key of the legitimate user, as such information is easy to obtain in practice.

To initiate the attack, the adversary first attempted to log in as a legitimate user of the OpenSSH server from a remote client using public/private keys. This step, however, is merely to create a window to conduct row hammer attacks against the `sshd` process, which is created by the `sshd` service daemon upon receiving login requests. By receiving the correct public key for the legitimate user, the server tries to locate the public key in the lo-

<pre>callq pam_authenticate test %eax, %eax jne <error_handling></pre>	<pre>mov \$0, %eax test %eax, %eax jne <error_handling></pre>
--	---

(a) Code before attacks. (b) Code after attacks.

Figure 14: Pseudo code to illustrate attacks against the OpenSSH server.

cal file (`~/ .ssh/authorized_keys`) and, if a match is found, a challenge encrypted by the public key is sent to the client. Then the OpenSSH server awaits the client to decrypt his encrypted private key file and then use the private key to decrypt the challenge and send a response back to the server. In our attack, the adversary paused on this step while he instructed the DomU attacker VM to conduct the cross-VM row hammer attacks to obtain access to the physical memory of Dom0. The steps to conduct the row hammer attacks were the same as described in the previous paragraphs. Particularly, here the adversary searched for a piece of binary code of `sshd`—a code snippet in the `sshpam_auth_passwd()` function. The signature can be extracted from offline binary disassembling as we assume the binary code of the OpenSSH server is also available to the adversary.

Once the signature was found, the adversary immediately replaced a five-byte instruction “`0xe8 0x1b 0x74 0xfd 0xff`” (binary code for “`callq pam_authenticate`”) with another five-byte instruction “`0xb8 0x00 0x00 0x00 0x00`” (binary code for “`mov $0 %eax`”). Note here even though the memory page is read-only in the victim VM, Dom0, the adversary may have arbitrary read/write access to it without any restriction. Then the code snippet will be changed from Figure 14a to Figure 14b. Upon successful authentication, `pam_authenticate()` will return 0 in register `%eax`. The modified code assigned `%eax` value 0 directly, without calling `pam_authenticate()`, so the authentication will be bypassed.

Then the adversary resumed the login process by entering password to decrypt the private key. The private key was incorrect so this step would fail anyway. Then password authentication would be used as a fallback authentication method, in which the adversary can log in the server with any password, because it was not really checked by the server.

Again, the time to complete the OpenSSH attack depends on the number of physical pages scanned before meeting the targeted one. If the target physical page is the first to be examined by the adversary, the average time to complete the attack was 0.322s, which included the time to manipulate page tables, conduct row hammer attacks to induce the desired bit flip, search the target page for specific patterns, and inject code in the target

memory. If additional memory pages need to be scanned, the average time to complete the pattern recognition in a 4KB memory page was 58μs.

We note the two examples only illustrate some basic uses of our presented cross-VM row hammer attacks as attack vectors. Other innovative attacks can be enabled by the same techniques. We leave the exploration of other interesting attacks as future work.

6.4 Prevalence of Xen PVM in Public Clouds

As shown in prior sections, Xen PVMs (paravirtualized VMs) are very vulnerable to privilege escalation attacks due to row hammer vulnerabilities. However, they are still widely used in public clouds. Amazon EC2⁷ as a leading cloud provider still offer PV guests in many of its instance types (see Table 3). Other popular cloud providers such as Rackspace⁸ and IBM Softlayer⁹ are also heavily relying on PV guests in their public cloud services. In addition, PVMs are also the primary virtualization substrate in free academic clouds like Cloudlab¹⁰.

The prevalence of PV guests provides adversaries opportunities to perform bit detection, and hence double-sided row hammer attacks in public clouds. With detected bit flips, it also allows malicious page table manipulation to enable arbitrary cross-VM memory accesses. This kind of hardware attack is beyond control of the hypervisor. Victims will suffer from direct impairment of the system integrity or more sophisticated exploits of the vulnerability from attackers.

cloud	instance types
Amazon EC2 [7]	t1, m1, m2, m3, c1, c3, hs1
Rackspace [28]	General purpose, Standard
Softlayer	Single/Multi-tenant Virtual Server
Cloudlab	d430, d810, d820, C220M4, C220M4, c8220(x), r320, dl360

Table 3: Prevalence of Xen paravirtualized VMs in public clouds.

7 Discussion on Existing Countermeasures

In this section, we discuss the existing software and hardware countermeasures against the demonstrated cross-VM row hammer attacks.

⁷<https://aws.amazon.com/ec2/>

⁸<https://www.rackspace.com/>

⁹<https://www.softlayer.com/>

¹⁰<https://www.cloudlab.us/>

Row hammer resistance with hardware-assisted virtualization.

Many of the attacks presented in this paper (*e.g.*, bit detection, double-sided row hammering, and also cross-VM memory accesses enabled by page table manipulation) require the adversary to know the machine address of his virtual memory. One way to prevent physical address disclosure to guest VMs is to adopt hardware-assisted virtualization, such as Intel’s VT-x [31] and AMD’s AMD-V [2]. Particularly, VT-x employs Extended Page Tables and AMD-V introduces Nested Page Tables [1] to accelerate the processor’s accesses to two layers of page tables, one controlled by the guest VM and the other controlled by the hypervisor. In this way, the guest VMs may no longer observe the real physical addresses, as they are not embedded in the PTEs any more. Hardware-assisted virtualization also prevents direct manipulation of page tables, and thus the privilege escalation attacks presented in this paper are not feasible.

The transition from Xen paravirtualization to hardware-assisted virtualization in public clouds started a few years ago, but the progress has been very slow. One reason is that paravirtualization used to have better performance than hardware-assisted virtualization in terms of networking and storage [9]. However, with the recent advances in hardware-assisted virtualization technology, some HVM-based cloud instances (*especially PV on HVM*) are considered having comparable, if not better, performance [7]. Even so, given the prevalence of paravirtualization in public clouds as of today, we anticipate it will take many years before such technology can gradually phase out. We hope our study offers to the community motivation to accelerate such trends.

Row hammer resistance with ECC-enabled DRAMs. As discussed in Section 2, the most commonly implemented ECC mechanism is single error-correction, double error-detection. Therefore, it can correct only one single-bit of errors within a 64-bit memory block, and detect (but not correct) 2-bit errors, causing the machines to crash. ECC memory will make the row hammer attacks much harder. Because 1-bit error and 2-bit errors are more common than multi-bit errors (*e.g.*, see Figure 13c), and it is very likely the privilege escalation attack will be thwarted either by bit correction or machine crashes before it succeeds. However, ECC memory does not offer strong security guarantees against row hammer attacks¹¹. It is still possible for an adversary to trigger multiple (> 3) bit flips in the same 64-bit word so that errors can be silently induced and later exploited. Particularly, if the true physical address of an extremely vulnerable rows is known to the adversary, hammering around this specific row will greatly increase the adver-

sary’s chances of success.

We believe a combination of hardware and software based defense will offer better security against row hammer attacks. On the one hand, hardware protection raises the bar of conducting row hammer attacks, and on the other hand, software isolation prevents successful exploitation once such vulnerability is found by the adversary.

8 Conclusion

In conclusion, we explored in this paper row hammer attacks in the cross-VM settings, and successfully demonstrated software attacks that exploit row hammer vulnerabilities to break memory isolation in virtualization. Many techniques presented in this paper are novel: Our graph-based bit detection algorithm can reliably determine row bits and XOR-schemes that are used to determine bank bits within one or two minutes. This novel method enables the construction of double-sided attacks, which significantly improves the fidelity of the attacks. The page table replacement attacks present a deterministic exploitation of row hammer vulnerabilities. The two examples we demonstrated in the paper, private key exfiltration from an HTTPS web server and code injection to bypass password authentication on an OpenSSH server, illustrate the power of the presented cross-VM row hammer attacks. The high-level takeaway message from this paper can be summarized as: (1) Row hammer attacks can be constructed to effectively induce bit flips in vulnerable memory chips, and (2) cross-VM exploitation of row hammer vulnerabilities enables a wide range of security attacks. We also believe that although server-grade processors and memory chips are more expensive and in contrast are less vulnerable to row hammer attacks, security guarantees needs to be achieved by both hardware and software solutions.

Acknowledgments

This work was supported in part by grant CRII-1566444 and CCF-1253933 from the National Science Foundation. The authors would like to thank the shepherd of our paper, Felix Schuster, and the anonymous reviewers for the constructive suggestions that greatly helped us improve the paper. We are grateful to CloudLab for providing us access to their servers.

References

- [1] AMD-V nested paging. [http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf](http://developer.amd.com.wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf). Accessed: 2016-06.

¹¹A recent study by Mark Lanteigne has reported that ECC-equipped machines are also susceptible to row hammer attacks [24].

- [2] AMD64 architecture programmers manual, volume 2: System programming. http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf. Accessed: 2016-06.
- [3] BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors. <http://support.amd.com/TechDocs/26094.pdf>. revision:3.30, issue date: 2016-02.
- [4] Exploiting the DRAM rowhammer bug to gain kernel privileges. <http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html>. Accessed: 2016-01-23.
- [5] How physical addresses map to rows and banks in DRAM. <http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html>. Accessed: 2016-01-30.
- [6] Intel 64 and IA-32 architectures software developers manual, combined volumes:1,2A,2B,2C,3A,3B and 3C. <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>. version 052, retrieved on Dec 25, 2015.
- [7] Linux AMI virtualization types. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/virtualization_types.html. Accessed: 2016-06.
- [8] Product Documentation for Red Hat Enterprise Linux. <https://access.redhat.com/documentation/en/red-hat-enterprise-linux/>. Accessed: 2016-06.
- [9] PV on HVM. http://wiki.xen.org/wiki/PV_on_HVM. Accessed: 2016-06.
- [10] Research report on using JIT to trigger rowhammer. <http://xlab.tencent.com/en/2015/06/09/Research-report-on-using-JIT-to-trigger-RowHammer>. Accessed: 2016-01-30.
- [11] X86 paravirtualised memory management. http://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Management. Accessed: 2016-01-23.
- [12] AICHINGER, B. P. DDR memory errors caused by row hammer. http://www.memcon.com/pdfs/proceedings2015/SAT104_FuturePlus.pdf.
- [13] BAINS, K., HALBERT, J. B., MOZAK, C. P., SCHOENBORN, T. Z., AND GREENFIELD, Z. Row hammer refresh command. US9236110, Jan 03 2014.
- [14] BAINS, K. S., AND HALBERT, J. B. Distributed row hammer tracking. US20140095780, Apr 03 2014.
- [15] BAINS, K. S., HALBERT, J. B., SAH, S., AND GREENFIELD, Z. Method, apparatus and system for providing a memory refresh. US9030903, May 27 2014.
- [16] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *37nd IEEE Symposium on Security and Privacy* (2016), IEEE Press.
- [17] CHISNALL, D. *The Definitive Guide to the Xen Hypervisor* (*Prentice Hall Open Source Software Development Series*). Prentice Hall PTR, 2007.
- [18] DONG, Y., LI, S., MALLICK, A., NAKAJIMA, J., TIAN, K., XU, X., YANG, F., AND YU, W. Extending Xen with intel virtualization technology. *Intel Technology Journal* 10, 3 (2006), 193–203.
- [19] GREENFIELD, Z., BAINS, K. S., SCHOENBORN, T. Z., MOZAK, C. P., AND HALBERT, J. B. Row hammer condition monitoring. US patent US8938573, Jan 30 2014.
- [20] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment* (2016).
- [21] JAHAGIRDAR, S., GEORGE, V., SODHI, I., AND WELLS, R. Power management of the third generation Intel Core micro architecture formerly codenamed Ivy Bridge. http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf, 2012.
- [22] KIM, D.-H., NAIR, P., AND QURESHI, M. Architectural support for mitigating row hammering in DRAM memories. *Computer Architecture Letters* 14, 1 (Jan 2015), 9–12.
- [23] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *41st Annual International Symposium on Computer Architecture* (2014), IEEE Press.
- [24] LANTEIGNE, M. How rowhammer could be used to exploit weaknesses in computer hardware. <http://www.thirdio.com/rowhammer.pdf>, 2016. Accessed: Jun. 2016.
- [25] LIN, W.-F., REINHARDT, S., AND BURGER, D. Reducing DRAM latencies with an integrated memory hierarchy design. In *7th International Symposium on High-Performance Computer Architecture* (2001).
- [26] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy* (2015), IEEE Press.
- [27] MOSCIBRODA, T., AND MUTLU, O. Memory performance attacks: Denial of memory service in multi-core systems. In *16th USENIX Security Symposium* (2007), USENIX Association.
- [28] NOLLER, J. Welcome to performance cloud servers; have some benchmarks. <https://developer.rackspace.com/blog/welcome-to-performance-cloud-servers-have-some-benchmarks>, 2013. Accessed: Jun. 2016.
- [29] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX Security Symposium* (2016), USENIX Association.
- [30] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *16th ACM conference on Computer and communications security* (2009), ACM.
- [31] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C. M., ANDERSON, A. V., BENNETT, S. M., KAGI, A., LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer* 38, 5 (May 2005), 48–56.
- [32] VARADARAJAN, V., ZHANG, Y., RISTENPART, T., AND SWIFT, M. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium* (2015), USENIX Association.
- [33] WANG, D. T. *Modern Dram Memory Systems: Performance Analysis and Scheduling Algorithm*. PhD thesis, College Park, MD, USA, 2005.

PIkit : A New Kernel-Independent Processor-Interconnect Rootkit

Wonjun Song, Hyunwoo Choi, Junhong Kim, Eunsoo Kim, Yongdae Kim, John Kim

KAIST

Daejeon, Korea

{iamwonjunsong, zemisolsol, jh15, hahah, yongdaek, jjk12}@kaist.ac.kr

Abstract

The goal of rootkit is often to hide malicious software running on a compromised machine. While there has been significant amount of research done on different rootkits, we describe a new type of rootkit that is *kernel-independent* – i.e., no aspect of the kernel is modified and no code is added to the kernel address space to install the rootkit. In this work, we present *PIkit* – Processor-Interconnect rootkit that exploits the vulnerable hardware features within multi-socket servers that are commonly used in datacenters and high-performance computing. In particular, PIkit exploits the DRAM address mapping table structure that determines the destination node of a memory request packet in the processor-interconnect. By modifying this mapping table appropriately, PIkit enables access to victim’s memory address region without proper permission. Once PIkit is installed, only user-level code or payload is needed to carry out malicious activities. The malicious payload mostly consists of memory read and/or write instructions that appear like “normal” user-space memory accesses and it becomes very difficult to detect such malicious payload. We describe the design and implementation of PIkit on both an AMD and an Intel x86 multi-socket servers that are commonly used. We discuss different malicious activities possible with PIkit and limitations of PIkit, as well as possible software and hardware solutions to PIkit.

1 Introduction

Rootkits are used by attackers for malicious activities on compromised machines by running software without being detected [47]. Different types of rootkits can be installed at the application-level, kernel-level, boot loader level, or hypervisor level. There has been significant amount of research done on different types of rootkits [57, 28, 25, 24] as well as different rootkit detections [21, 31, 11]. Recently, there have been other types of rootkits [20, 50] that exploit vulnerable hardware features such as de-synchronized TLB structures and un-

		Malicious Payload	
		User-Level	Kernel-Level
Vulnerabilities	Software	t0rn [35], lrk5 [48], dica [23], etc.	ROR [25], DKOM [15], knark [17], etc.
	Hardware	This work (PIkit)	Cloaker [20] Shadow Walker [50]

Table 1: Classification of different rootkit attacks.

used interrupt vector. Prior work on rootkit can be classified based on whether the payload consists of user-level or kernel-level code and whether rootkit is installed in the software or with the support of the hardware (Table 1). In this work, we propose a new type of rootkit that modifies hardware state but enables malicious activities with simple user-level code that consists of read/write memory accesses to user-level memory space. While prior work on user-level software rootkit often modified existing files [35, 48, 23], this work does not modify the kernel or any existing files. Since this work does not require any code modification or code injection to the kernel, traditional approaches to detect software rootkit, such as kernel integrity monitoring [42, 31, 53] or code signature-based detection [1, 2] can not be used for detection.

In this work, we present PIkit, processor-interconnect rootkit, that exploits hardware vulnerability in x86 multi-socket servers. x86 is the most dominant server processor in datacenters and high-performance computing [36] and a recent survey found over 80% of the x86 servers are *multi-socket servers*. The multi-socket servers contain a processor-interconnect that connects the sockets together (e.g., Intel QPI [37], AMD Hypertransport [10]) and we exploit the processor-interconnect to implement PIkit. Once PIkit is installed, the payload or the malicious code to carry out an attack exists in user space and appears like a “normal” user program – i.e., all of the memory accesses from the payload are legal memory accesses and it becomes very difficult to identify such user code as “malicious” code. As a result, PIkit is a seri-

ous threat to multi-socket servers that is difficult to detect with currently available rootkit detection mechanisms.

PIkit that we propose is implemented on x86 servers from both AMD and Intel to demonstrate how PIkit enables an attacker to continuously access the victim’s memory region without proper permission.¹ In particular, we exploit the configurability in the DRAM mapping table that enables a memory request packet to be routed to a different node by modifying a packet’s destination node. We also exploit the extra entries available in the DRAM mapping table to define an *attack memory region* when installing PIkit. As a result, user-level memory read or write requests to the attack memory region get re-routed to another memory region or the *victim’s memory region*. To the best of our knowledge, this represents the first rootkit where with the support of hardware state modification, user-level code or payload is sufficient to carry out malicious activities.

Most rootkits often modify some components of the OS while other rootkits add malicious payload to the kernel without modifying the OS to carry out malicious activities. However, such approaches can be exposed by signature-based detection and integrity checking. In comparison, PIkit only requires user-level payloads with the support of hardware state modifications as no malicious payloads to the kernel space are added or modified (Figure 1). In addition, any signature scan of the memory that contains the user-level payload can not identify the user code as “malicious” since the memory accesses appear to be legal accesses as the malicious access is only achieved through the support of the hardware modifications. As a result, PIkit demonstrates how a very stealthy rootkit can be achieved compared to previously proposed rootkits.

The proposed PIkit is a non-persistent rootkit [20, 45, 41] and does not remain after the server is restarted. However, servers are rarely rebooted to minimize the impact on availability – for example, one study measured the average time between reboot in the server room to be 481 days [22]. Thus, PIkit poses a serious threat to servers while powered on. Prior work on non-persistent rootkit [20] has argued that non-persistence can also significantly reduce the detectability of rootkits.

In particular, the contributions of this work include the followings.

- We show that the DRAM address mapping table structure in the processor-interconnect of multi-socket servers has security vulnerability that can be exploited maliciously in both an AMD and an Intel-based x86 server.

¹This work can also be viewed as a “backdoor” since once PIkit is installed, it provides a covert mechanism for the attacker to gain privileged access to the system.

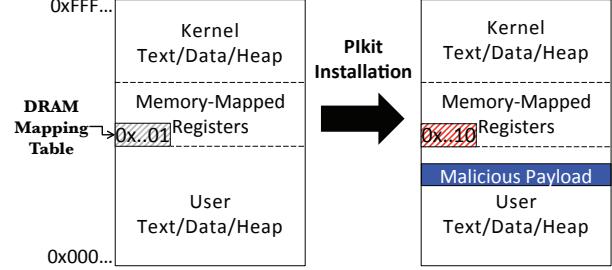


Figure 1: High-level overview of PIkit showing DRAM address mapping table modification and user-level malicious payload added for malicious activities.

- We describe a new type of rootkit that is *kernel-independent* that requires only hardware state modification with user-level payload, no modification or addition to the kernel is necessary. In particular, we present PIkit, Processor-Interconnect rootkit, that exploits the mapping table vulnerability to enable the malicious attacker privileged access (both read and write) to a victim memory address region with only user-level access.
- Once PIkit is installed, we demonstrate how different malicious activities can be carried out including bash shell credential object attack, shared library attack, and keyboard buffer attack with only user-level memory accesses.
- We describe alternative solutions, including a software-based, short-term solution to detect PIkit as well as hardware-based, long-term solutions to prevent PIkit.

We responsibly disclosed this vulnerability to CERT before publishing this paper. The rest of the paper is organized as follows. We first describe our threat model in Section 2 and background into processor-interconnect as well as related work. The DRAM address mapping table structure is described and analyzed in Section 3. The design and implementation of PIkit that modifies the mapping table structure is described in Section 4 and we illustrate different malicious activities in Section 5. We provide some discussion on different solutions as well as limitations of PIkit in Section 6 and we conclude in Section 7.

2 Background

2.1 Threat Model

In this work, we assume an attacker and a victim share the same multi-socket server that is commonly used in cloud servers and high-performance computing. We assume an attacker has no physical access to the hardware and also assume the same threat model as prior work on rootkit attack – the attacker, through some vulnerabilities (e.g., vulnerabilities in commodity OSes [39, 16, 6] or perhaps through an administrator (or an insider) who

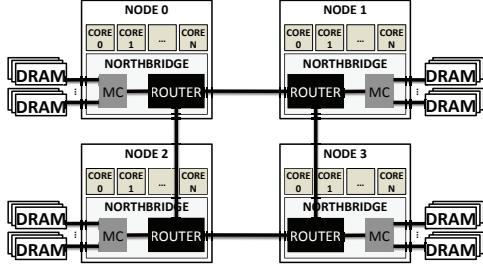


Figure 2: Block diagram of a processor-interconnect in a 4-socket server. (MC: memory controller)

maliciously provides one-time root access [54]) or social engineering, is assumed to have privileged access for rootkit installation. Once this is achieved, the next goal is to avoid detection of intrusion while carrying out malicious activity, similar to other rootkits. After PIkit is installed, it becomes very difficult to detect or determine the source of the attack as there are no changes to the kernel in the target system.

2.2 Processor-Interconnect Overview

A high-level block diagram of a processor-interconnect is shown in Figure 2. The processor-interconnect provides connectivity between multiple nodes (or sockets) in a NUMA (non-uniform memory access) server with each node containing multiple cores. The router within the Northbridge is used to connect to other nodes and it is also used to access local memory. For simplicity, the example in Figure 2 shows a ring topology to interconnect the 4 nodes together but for a small-scale system, all of the nodes can be fully connected as well. Given the processor-interconnect and its topology, the routing algorithm determines the path taken by a packet to its destination [19]. To provide flexibility, a routing table is commonly used in the processor-interconnect to implement the routing algorithm. Based on the destination of the packet, a routing table look-up is done to determine the appropriate router output port that the packet needs to be routed through. However, the routing table is only used to determine which router output port should be used (and the routing path); the routing table is not responsible for determining the packet destination.

The packet destination information is determined by the packet header. The format of messages (or packets²) in the processor-interconnect is similar to other interconnection networks [19] as shown in Figure 3. A packet is the high-level message that is sent between the nodes, which can include memory requests, cache line replies, coherence message, etc. A packet is often partitioned into one or more *flits* or flow control units within the interconnection networks – thus, a packet can be partitioned into a head flit, one or more body flits, and a

²A message can consist of multiple packets but within the processor-interconnect, messages are often single packet.

tail flit. The head flit contains additional “header” information, which can include the packet type and both the source and destination node information of the packet. The body/tail flits do not contain destination information but only the payload and simply follows the head flit from the source to its destination.

2.3 Related Work

To the best of our knowledge, very few prior research have investigated security vulnerabilities within the hardware of the processor-interconnect in multi-socket servers. Song et al. [49] demonstrated the security vulnerability of the routing table in a multi-socket server. This vulnerability enabled performance attacks by sending packets through longer routes and degrading both interconnect latency and bandwidth. In addition, it also enabled system attacks by creating a livelock in the network to crash the system. However, the routing table did not modify the *destination* of a packet and thus, the scope of the attack was limited. In this work, we show how the *destination* of a packet can be changed by modifying the DRAM mapping table to enable a rootkit attack.

Rootkit Attacks: User-level, software rootkits (upper left box in Table 1) often modify existing system utilities to enable malicious codes. Lrk5 [48], T0rn [35] and Dica [23] replace the system binaries (e.g., ls, ps and netstat) with modified versions to hide files, processes or network connections. SAdoor [40] is a non-listening daemon that grabs packets directly from the NIC and watches for special key and command packets before executing a pre-defined command (e.g., /bin/sh). However, it has been shown that these rootkit are often easily detected by integrity checking for the system binaries.

Traditional kernel-level, software rootkits (upper right box in Table 1) exploit the control hijacking and interception, modifying static kernel data structures (e.g., system call table) to jump to malicious codes indirectly. DKOM [15] introduced a more advanced kernel-level rootkit approach which exploits dynamic (non-control) kernel data structures (e.g., processor descriptors) to install the rootkit. Hofmann et al. [24] introduced a rootkit which allows for malicious control flows by replacing pointer variables. Hund et al. [25] introduced a return-oriented rootkit based on Return-Oriented Programming [14] to bypass integrity checking for the kernel code. However, these kernel-level, software rootkits require modifications to the kernel and can be detected with protecting return addresses on the stack and critical data structures from the modification.

In addition to software rootkits, hardware-supported rootkits have been proposed (lower right box in Table 1). ShadowWalker [50] hid the trace of the rootkit by hooking the page tables while Cloaker [20] exploited ARM-specific architectural feature to conceal the rootkit with-



Figure 3: Packet format in interconnection networks with packet consisting of multiple flits.

out altering existing kernel code. However, these work require adding malicious payload to the kernel which can be prevented by guaranteeing the execution of only verified kernel code or detected by checking the flow of the hijacked code. In comparison, while PIkit also leverages a hardware-vulnerability, PIkit enable malicious activity with only user-level code.

The rootkits that we categorized in Table 1 and described earlier focus mostly on the software or hardware (CPU)-related rootkits. There have also been other device-specific rootkits such as network interface card [51], hard-drives [57], USB mouse [33], and printers [18]. In addition, rootkits involving BIOS [56] have also been proposed. However, these rootkits also involve modifying existing firmware to carry out the attack. Subvirt [28] presented a more stealthy rootkit by using virtual machine monitor (VMM) but Subvirt can also be detected with physical memory signature scans. Run-DMA [44] is a DMA (direct memory access) rootkit attack that enables a “malicious computation” where attacker modifies data inputs to induce arbitrary computation, in comparison to a more traditional “malicious code” model attack. The PIkit that we present in this work is not necessarily limited to either malicious computation or malicious code model since any region in the memory can be modified as long as the memory mapping can be determined by the attacker.

Rootkit Detection: The rootkit detection can be largely divided into two types, checking the integrity of kernel codes and data structures and detecting malicious control flows (e.g., hooking system call table and interrupt vector table). Copilot [42] detects the modification of kernel and jump tables with the separated PCI card monitor. However, such approaches do not guarantee hardware register integrity, such as the register modified in Cloaker [20] or the DRAM address mapping table modified in this work. KI-mon [31] introduced a hardware-assisted monitor, which snoops all bus traffic to verify updates of the kernel objects with the address filter while Shark [53] proposed an architectural supported rootkit monitor. However, since PIkit does not access the kernel objects directly, such approaches cannot detect PIkit.

3 Analysis of Processor-Interconnect in Multi-Socket Servers

In this section, we describe the *DRAM address mapping table* structure within the Northbridge (or the un-

	Base Address	Limit Address	Destination ID
0	0x0000000000	0x041F000000	0
1	0x0420000000	0x081F000000	1
2	0x0820000000	0x0C1F000000	2
3	0x0C20000000	0x101F000000	3
4	RESERVED	RESERVED	RESERVED
5	RESERVED	RESERVED	RESERVED
6	RESERVED	RESERVED	RESERVED
7	RESERVED	RESERVED	RESERVED

Figure 4: An example of DRAM address mapping table for a 4-node system.

core) that we analyze in detail and then describe how it can be exploited to enable a hardware vulnerability-based rootkit attack through the processor-interconnect.

3.1 DRAM Address Mapping Table

One of the critical information in packet’s header is the destination information; this information is often based on the destination memory address in modern multi-socket servers. The destination node is determined by a memory address mapping table structure between the core and the processor-interconnect router – we refer to this as the *DRAM Address Mapping Table*.³ Based on the destination address of the packet, the DRAM address mapping table determines the destination. As a result, regardless of the address, the packet is simply forwarded to the destination based on the packet header information within the processor-interconnect.

A separate copy of DRAM address mapping table structure exists within each node of a multi-socket system, between the core (or the last level cache) and the router. Each entry in the mapping table contains a DRAM physical memory address range, often including the start (or the base) address and the limit address. Each entry also contains the destination node information – thus, if an address falls within the address range, the destination node information is appended to the packet. An example of a DRAM address mapping table is shown in Figure 4.

The number of entries in the DRAM address mapping table should be equal to or greater than the maximum number of nodes in the system. In the AMD system that we evaluate (AMD Opteron 6128), the system contains 4 nodes but the system is scalable up to 8 nodes. Thus, the DRAM mapping table contains 8 entries with only 4 of the entries used and the remaining 4 entries not used (or shown as RESERVED in Figure 4).

The DRAM address mapping table is initialized by the BIOS at boot time. Since the table entries are memory-mapped registers, the BIOS uses memory operations to initialize the memory mapping table. The contents of

³In Intel-based NUMA systems, this structure is referred to as the DRAM address decoder [5] while in AMD- multi-socket systems, similar structures are referred to as DRAM address map register [13].

the address mapping table entries are dependent on the DRAM capacity installed on each node. To determine the address range for each entry (and each node), the BIOS calculates the current memory capacity by obtaining DRAM information such as the number of rows, banks and ranks from the SPD (Serial Presence Detect) [4] on the DRAM.

3.2 Vulnerable Hardware Features

To implement PIkit, we exploit the following three aspects of the DRAM address mapping table in multi-socket servers.

Configurability: The memory mapping table needs to be configurable since the memory capacity per system (and per node) is flexible and determined by the system user.

Extra entries: Since the system needs to be designed for scalability, the number of entries in the DRAM address mapping table needs to equal to or greater than the largest system configuration. For most multi-socket servers today, the maximum number of nodes in the system is often 8; however, the most dominant NUMA servers on the market are often 2 or 4 nodes [26] and thus, there are memory mapping table entries that are unused.

Discrepancy: The DRAM mapping table content values can be modified after the initialization such that the values are not consistent with the original values. This discrepancy may or may not cause a problem, depending on how the table is modified.⁴

Thus, given these three hardware vulnerabilities, *the destination node information of packets in the processor-interconnect can be modified such that the packets are sent to a different node (and its corresponding victim's memory address) to allow an attacker to access unauthorized memory space without proper permission*. In the following section, we describe the challenges in the design and implementation of PIkit.

4 PIkit Design & Implementation

PIkit installation procedure that includes modifying the DRAM address mapping table is described in this section. We first provide an overview and describe how the attack address region needs to be prepared by the attacker, and then modify the DRAM mapping table based on the attack address region. Our initial design and implementation are shown for an AMD-based server but we also discuss how PIkit can be implemented on Intel-based servers as well.

⁴Steps to ensure that the discrepancy does not cause a problem is discussed in Section 4.3 (for AMD) and Section 4.5 (for Intel).

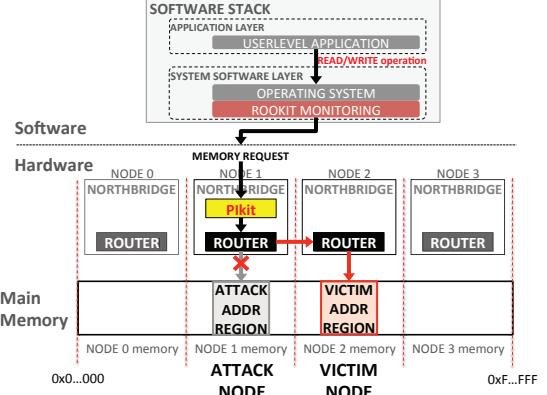


Figure 5: High-level description of the proposed PIkit on a 4-node multi-socket server.

4.1 Overview

In this work, we define the *attack node* as the node in the multi-socket system where the DRAM address mapping table is modified to implement PIkit, and the *victim node* is the node where its memory is maliciously accessed, through user-level read or write operations without proper permission. The address region that is modified in the memory mapping table in the attack node is defined as the *attack address region* while the corresponding address in the victim node is the *victim address region*, as shown in Figure 5. The PIkit and the modification in the DRAM address mapping table result in read/write memory requests to the attack address region being routed to the victim address region. The high-level diagram in Figure 5 also shows how the PIkit relates to the entire system. Most rootkit monitoring mechanisms (or solutions) exist at the software-level but the PIkit that we propose in this work is at the hardware-level (within the processor-interconnect) and exploiting vulnerability in the mapping table structure.

After the core injects a memory request into the Northbridge and before the packet is actually routed through the processor-interconnect by the router, a packet header is created based on the physical address of the memory request as shown in Figure 6. The processor-interconnect does not observe the memory address as that is included in the packet payload⁵ and is only observed at the destination (i.e., memory controller). The processor-interconnect only observes the destination node information that is appended at the interface between the core and the router. The PIkit that we propose in this work exploits this vulnerability of modifying the *destination* of a packet – in particular, the DRAM address mapping table structure to modify the packet’s destination.

⁵Packet payload refers to the non-header or the data portion of the packet while the payload terminology used in the rest of this paper refers to the malicious code used after rootkit is installed.

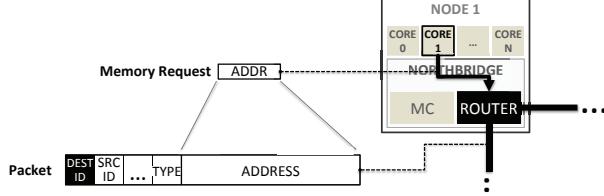


Figure 6: Hardware overview of memory request and packet header.

4.2 Defining Attack Address Region

Before the PIkit installation, the attacker first needs to prepare the attack address region such that only the attacker has access to that particular memory region. This step is critical to ensure that PIkit and the DRAM address mapping table modification do not cause any unknown system behavior including crashing the system. For example, if another program (or user) attempts to access the attack memory region after the PIkit is installed, the memory access will be routed to the victim node and unexpected system behavior will occur if a memory write is being done on an unintended memory address – i.e., it can cause critical data in kernel space to be overwritten.

The memory address range of the attack address region needs to be equal to granularity or the *resolution* of the memory mapping table. Although each entry in the DRAM address mapping table specifies both the base and the limit addresses, the full width of the address (i.e., 48 bits) is not stored as some of the lower bits are not specified in the DRAM address mapping table. For example, in the AMD Opteron 6128 system that we evaluate, the granularity of the memory mapping table is 16 MBs as only 24 most-significant bits are stored – thus, the attacker needs to obtain at least 16 MBs of physically contiguous memory region. To achieve this, we take advantage of huge pages that are commonly available. In the system that we evaluate, we used 1 GB huge page. After successful `malloc` for a huge page, an address range within the contiguous memory region allocated can be used as the attack address region, as long as the process that received the memory allocation is continuously running.

4.3 Modifying the DRAM Address Mapping Table

The DRAM address mapping table consists of physical addresses while the attack address region obtained in the previous section are virtual addresses. Thus, an attacker needs to obtain the translated physical address of the attack region before modifying the DRAM address mapping table. The translation of the virtual to physical address can be determined by using `/proc/(pid)/pagemap` interface from the Linux kernel. Based on this translation information, the DRAM address mapping table can be modified using the corresponding physical address of

	Base Address	Limit Address	Destination ID
0	0x0000000000	0x041F000000	0
1	RESERVED	RESERVED	RESERVED
2	0x0820000000	0x0C1F000000	2
3	0x0C20000000	0x101F000000	3
4	0x0420000000	0x07BF000000	1
5	0x07C0000000	0x07C1000000	2
6	0x07C2000000	0x081F000000	1
7	RESERVED	RESERVED	RESERVED

Figure 7: A modified DRAM address mapping table where entry 5 (highlighted) is used as the attack address region.

the attack region.

An example of a modified mapping table is shown in Figure 7, based on the original DRAM address mapping table shown earlier in Figure 4. We assume the attack node is node 1 and the victim node is node 2, with the attack address region defined as the address between 0x07C0000000 and 0x07C1000000 in node 1. In Figure 7, the entry 1 of the table which originally identified node 1 memory region has been removed. Instead, the same address range has been partitioned across entries 4, 5, and 6 of the modified DRAM mapping table (Figure 7). The key difference compared with the original mapping table is that for entry 5, the destination node ID has been modified such that it is no longer node 1 but modified to node 2 – thus, entry 5 represents the *attack address region*. Any address requests between 0x07C0000000 and 0x07C1000000 from node 1 have a destination of node 2 added to the packet header, instead of the original destination of node 1. When this particular packet arrives at node 2, the DRAM memory controller within the node 2 will receive this packet and convert the address within the payload of the packet into the actual *victim address region*. For example, in the AMD system that we evaluate, address 0x07C0000000 from node 1 ends up being mapped to address 0x0840000000 in node 2. As a result of the mapping table modification, the physical memory connected to node 1 that originally corresponded to the address range between 0x07C0000000 and 0x07C1000000 can no longer be physically accessed from node 1.

Since the table entries are memory-mapped registers, the entries can be modified through system read/write commands (e.g., `setpci` utility). However, to properly modify the DRAM address mapping table entries, the following caution must be taken.

1. The new entries must be written before the old entry is removed (e.g., in Figure 7, entries 4, 5, 6 must be written before entry 1 is cleared).
2. For the new entries added, the base address register must be written before the limit address register.
3. For the existing old entry that needs to be removed, the limit address register must be cleared first, before the base address register.

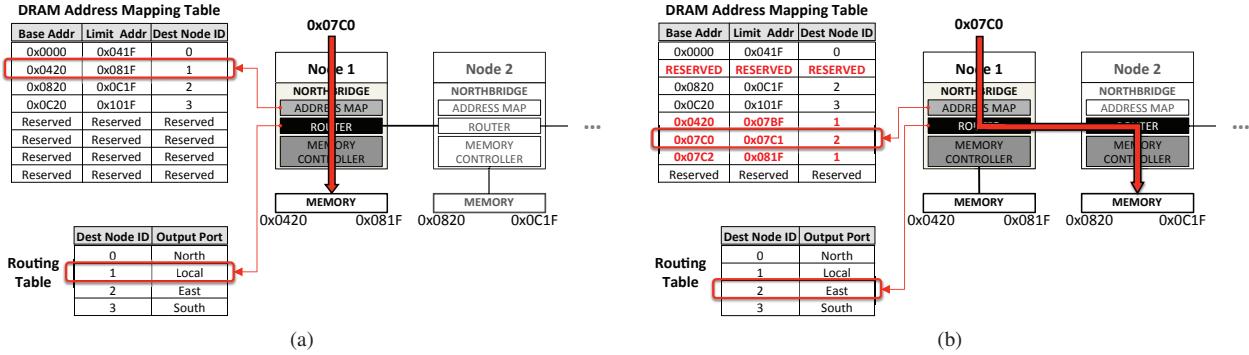


Figure 8: Example of PIkit (a) before and (b) after PIkit is installed on an AMD Opteron 6128 server.

Since memory accesses continuously occur in the system, randomly changing the mapping table in any order can result in a memory request that is not able to match an entry in the mapping table and result in a system crash.

4.4 Example

A complete example of PIkit is shown in Figure 8 for a 4-node system with only 2 nodes shown in the figure for simplicity. The same DRAM address mapping tables shown earlier in Figure 4 and Figure 7 are used in the example.⁶ Assume a read access to address 0x07C0000000 is made by a core in node 1. With an unmodified DRAM address mapping table, the address 0x07C0000000 finds a match in entry 1 and determines that the destination should be node 1. Based on this information, a routing table look-up is done for the node 1 to determine the output port. Since the current node is the local node, the output port determined by the routing table is the “Local” port and the memory request is routed appropriately to the local node’s memory.

However, with the modified DRAM address mapping table (Figure 8(b)), the same request to address 0x07C0000000 finds a match to entry 5 where the destination is now node 2. Based on this new destination node ID information, the routing table look-up is done within the router and the output port returned is the “East” port – thus, the packet is routed to node 2. Within the node 2, the packet is simply treated as a packet that was destined for node 2 (or the “Local” output port) and will be routed to the memory controller. Since the processor-interconnect only looks at the destination node information to determine where to send the packet, the PIkit shown in Figure 8(b) results in packets accessing a memory region where it does not have proper permission. If this packet was a read request, the packet would read data from the corresponding victim memory address while if this packet was a write request, the packet would modify or overwrite existing data in the victim memory address.

⁶For simplicity, the lower 3 bytes of the address are not shown in Figure 8.

DRAM Address Mapping Table		
Base Addr	Limit Addr	Dest Node ID
0x0000	0x041F	0
0x0420	0x081F	1
0x0820	0xC1F	2
0x0C20	0x101F	3
Reserved	Reserved	Reserved

Routing Table	
Dest Node ID	Output Port
0	North
1	Local
2	East
3	South

DRAM Address Mapping Table		
Base Addr	Limit Addr	Dest Node ID
0x0000	0x041F	0
RESERVED	RESERVED	RESERVED
0x0820	0xC1F	2
0x0C20	0x101F	3
0x0420	0x07BF	1
0x07C0	0x07C1	2
0x07C2	0x081F	1
Reserved	Reserved	Reserved

Routing Table	
Dest Node ID	Output Port
0	North
1	Local
2	East
3	South

Figure 9: Example of how the Source Address Decoder (SAD) can be modified on the attack node to implement PIkit on an Intel Sandybridge architecture.

4.5 Extending PIkit to Intel Architecture

In the previous sections, we described how PIkit is implemented on an AMD multi-socket server and in this section, we discuss how PIkit can be extended to an Intel-based server. A structure similar to the memory mapping table exists within Intel x86 server architecture and is referred to as the *Source Address Decoder (SAD)* [5]. A key difference with the AMD architecture in the memory mapping table is that instead of specifying both the base and the limit memory address for each entry, only the limit address is specified. In addition, a valid bit per entry exists in the SAD which specifies if the entry is enabled or not. As a result, the “base” address is implied from the previous entry limit address and PIkit design needs to properly add/modifies entries of SAD to ensure proper behavior for memory accesses. An example of how the SAD table can be modified is shown in Figure 9. The initial entries are first duplicated in the table (step (2)) and then, the initial entries are invalidated (step (3)) before the addresses are modified (step (4)) and then, the modified addresses are made valid (step (5)) to create an attack address region with entry 1 of the SAD table.

Another key difference in the Intel architecture compared with the AMD system is the *Target Address Decoder (TAD)* which is accessed before the address is sent to the memory controller at the destination node.⁷ TAD is an additional table that is responsible for mapping discontinuous address regions [5] and includes both a limit address and an offset. While the purpose of the “offset” within the TAD is to relocate the memory location as necessary, it enables PIkit to be implemented by defining

⁷While a similar structure existed in the AMD system that we evaluated, it only had a single entry and could not be exploited for PIkit.

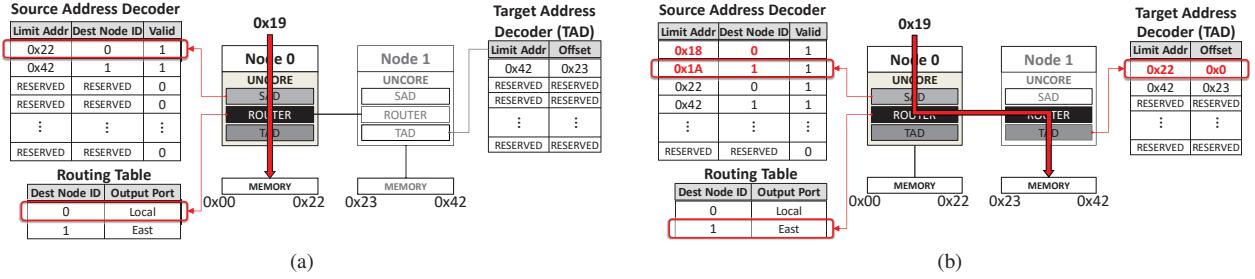


Figure 10: PIkit example on an Intel Sandybridge-based server (a) before and (b) after PIkit is implemented. For simplicity, the TAD on Node 0 and SAD 0 on Node 1 are not shown.

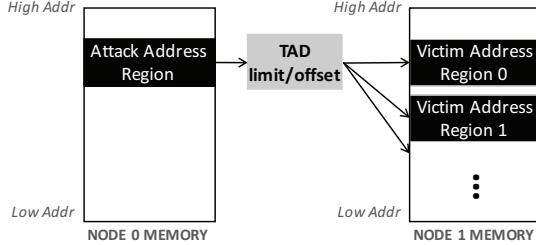


Figure 11: Impact of the Target Address Decoder (TAD) on the mapping of the attack memory region to the victim memory region.

the victim memory address region based on the attack address region. The impact of TAD offset is shown in Figure 11. Since the offset is subtracted from the address, by varying offset entry, the attack memory address region can be mapped to different victim memory region based on the offset value and enables a more “controlled” attack by providing control over the memory mapping (i.e., victim’s address range). One constraint is that since the offset is being subtracted, the victim memory address region can only be equal or smaller than the attack memory address region.

A PIkit example for an Intel-based architecture is shown in Figure 10 based on the SAD modification shown in Figure 9. The PIkit consists of both the SAD modification in the attack node and the TAD modification in the victim node. Thus, the same vulnerability that was exploited for PIkit on the AMD-based system is available in the Intel-based servers – the table structures are memory-mapped registers that are configurable and extra number of entries are available. Based on documentations [5], the number of entries for SAD is 20 and thus, it is more than sufficient for a 2-node multi-socket system.

5 Malicious User-level Payloads

After PIkit is installed, PIkit enables access to the victim address region regardless of the privilege level. In this section, we describe different malicious activities with user-level payloads that can exploit PIkit. While many different attacks (and payloads) have been proposed as part of rootkit attacks, previous attacks often require

Description	Value
System	AMD Opteron 6128
# of Sockets (Nodes)	2 (2 per socket)
# of Cores	4 per node
Interconnect	6.4 GT/s HT 3.0
# of HT Links	4 per node
OS version	Linux Kernel 3.6.0

Table 2: Dell PE R815 server used in our evaluation.

leveraging (or modifying) some OS capability or creating additional payload to mimic the OS. In comparison, the *malicious payload for PIkit is fundamentally different as the payload is relatively simple with the source code mostly consisting of memory read and write commands.*⁸ The main challenge with PIkit payload is determining the attack (or the corresponding victim) address region to carry out the malicious activity.

5.1 Bash Shell Credential Object Attack

In the operating system, a process is represented by a process control block (PCB) data structure in the privileged memory space. The process control block has critical information such as memory information, open-file lists, process contexts and priorities, etc. In particular, we exploit the credential kernel data structure which is contained within the PCB and is responsible for access controls of a process in the Linux kernel. If the attacker locates the credential data structure in the victim address region, the attacker can modify any value within the credential data structure with PIkit. In this work, we modify the UID or the EUID of a bash shell process to achieve root privilege escalation. An overview of the malicious activity is shown in Figure 12. We demonstrate this attack on a 4-node AMD server described in Table 2.

5.1.1 Scanning the Fingerprint

In this attack, we assume the attacker uses a common user-level application, *Bash Shell*, to obtain root privilege. After the PIkit is installed on the attack node, the attacker starts the bash shell on the *victim* node and attempt to modify the credential data structure for privi-

⁸Pseudo-code for the malicious payload is shown in Appendix for the three different malicious activities described in this section.

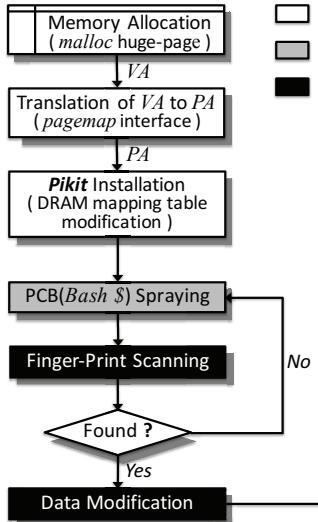


Figure 12: High-level overview of the attack with PIkit for privilege escalation.

lege escalation. One challenge before obtaining privilege escalation is determining the actual address of the PCB (or in particular, the credential data structure) of the bash shell on the victim node. In order to determine the memory location, we use the fingerprint of the credential data structure to identify the proper starting address. As shown in Figure 13, the credential data structure consists of multiple integer variables and pointers which contain 64 bits addresses. Since the bash shell user-level process is owned by the attacker, the attacker knows the user-/group ID of the process and can use the consecutive user and group IDs (e.g., UID, GUID, ..., FSGID), as shown in Figure 13 ① as these variables often have the same values.

To increase the accuracy of the fingerprint, additional pointer information within the credential data structures can also be used. Since most of the x86-64 Linux systems use the specific virtual address ranges for the kernel objects based on the kernel virtual address map [29] (e.g., 0xffff880000000000 – 0xffffc7ffffffffff for the direct mapping), the pointers shown in Figure 13 ② should have virtual addresses that are within this range. This approach is similar to what was used in prior work [46] that used virtual address characteristic to find the process control block used by Window operating system in dumped memory. In addition, some addresses of different variables used in kernel space are publicly available even in the user-level space, including the *Symbol Lookup Table* ('/boot/System.map') [32]. In particular, the virtual address of '*user_ns*', shown in Figure 13 ③, can be found in the *Symbol Lookup Table* which is determined at kernel compile time and can be used as part of the fingerprint.

Based on the three types of information described above, a fingerprint for the credential kernel data struc-

```

struct cred {
    ...
    uid_t      uid;          /* real UID of the task */
    gid_t      gid;          /* real GID of the task */
    uid_t      suid;         /* saved UID of the task */
    gid_t      sgid;         /* saved GID of the task */
    uid_t      euid;         /* effective UID of the task */
    gid_t      egid;         /* effective GID of the task */
    uid_t      fsuid;        /* UID for VFS ops */
    gid_t      fsgid;        /* GID for VFS ops */
};

struct key *thread_keyring; /* keyring private to this thread */
struct key *request_key_auth; /* assumed request_key authority */
struct thread_group_cred *tgcrcd; /* thread-group shared credentials */
...
struct user_namespace *user_ns; /* cached user->user_ns */
...

```

Figure 13: The credential kernel data structure in the Linux kernel 3.6.0 and the fingerprint that we exploit in this work, with the fingerprint highlighted with a rectangle.

ture can be used to determine the location of the credential data structure. The attacker from the attack node can issue read operations for the attack address region – determined by the allocated memory region and the modified DRAM address mapping table on the attack node as described in Section 4 – and begin the fingerprint scanning. The read requests will be routed to the victim node and the data in memory will be returned to the attack node where it will be compared against the fingerprint. If there is a match, the starting address of the credential data structure is found and the attacker can modify the data. If no match is found, the credential data structure of the bash shell is not found on the victim address region – thus, the attacker needs to stop the current bash shell process and restart the bash shell on the victim node, and repeat the fingerprint scanning.

Note that the attacker does not need to know the starting *virtual* address of the kernel data structure on the victim node. In fact, the appropriate *physical* address within the victim address region does not need to be known as well. Only the *corresponding* address on the attack node needs to be determined from the fingerprint scanning and the attack (or the modification of the data) is done based on the physical address of the attack node and the corresponding virtual address within the attack node is used in the actual data modification.

5.1.2 Modifying the Data

Once the corresponding address of the credential data structure is determined from the scanning, the offset within the data structure can be easily determined based on the credential data structure definition (e.g., Figure 13) and the different variables within the data structures can be easily modified with PIkit. Thus, the root privilege can be obtained by modifying either the *euid* (Effective User ID) or the *uid* (User ID) field within the credential data structure to a value of 0 (instead of the original value) as the *uid* of 0 specifies the root user. In order to overwrite this variable, a memory write instruction in assembly language can be used by the attacker to obtain the root privilege – e.g.,

```
movnti $0, (Virtual Address)
```

where the virtual address is the address determined from

```
[smith@server ~]$ id
uid=500(smith) gid=500(smith) groups=500(smith) context=unconfined_u:u
[smith@server ~]$ id
uid=500(smith) gid=500(smith) euid=0(root) egid=0(root) groups=0(root)
ined_t:s0-s0:c0.c1023
```

Figure 14: Screen capture from `id` command of privilege escalation, before the attack and after the attack.

Mhead (8 Bytes)	Keyboard Input Buffer (256 Bytes)
	<pre>union mhead { bits64_t mh_align; /* 8 */ struct { char mi_alloc; /* ISALLOC or ISFREE */ /* 1 */ ① char mi_index; /* index in nextfl */ /* 1 */ u_bits16_t mi_magic2; /* should be == MAGIC2 */ /* 2 */ ② u_bits32_t mi_nbytes; /* # of bytes allocated */ /* 4 */ ③ } minfo; };</pre>

Figure 15: Keyboard buffer data structure in the bash 4.3 and the fingerprint that we exploit in this work, with the fingerprint highlighted with a rectangle.

scanning and PIkit routes this write instruction to the victim node. However, we used a non-temporal SSE instruction in our evaluation in order to bypass the cache within the processor. If the write occurs to the cache within the attack node, the effect of the root privilege escalation can be delayed until write-back to the memory occurs.

The result from the attack is shown in Figure 14, consisting of the ID information before and after that attack using the `id` command from of the bash shell on the victim node. The EUID of *Bash Shell* in the victim node is modified to root user and thus, root privilege escalation is achieved.

5.1.3 Spraying the Process Control Block

As described earlier, the attack address region is mapped to some victim address region on the victim node through PIkit. As a result, all memory accesses to the attack address region on the attack node are constrained to some victim address region based on the DRAM physical mapping which is not known. As a result, if a user-level application (i.e., bash shell) executing on the victim node is not placed in the victim address region, an attacker can not access the kernel objects of the process. To increase the probability that the credential data structure can be found, the PCB can be sprayed across the victim node by executing multiple bash shells on the victim node. This increases the probability that one of the processes (and the corresponding PCB) is placed within the victim address region and reduces the amount of time it takes to achieve privilege escalation.

5.2 Bash Keyboard Buffer Attack

In this section, we describe how PIkit can be exploited to carry out an information leakage attack on another user’s bash shell and perform a *bash keyboard buffer monitoring* attack. Since no data modification is required, this attack can be classified as a read-only attack. When a

[Timestamp]	[Attack Phy Addr]	[Victim Phy Addr]	[KBD buffer]
03:34:59	0x1eeeb6800	0x41eeb6800	mysqladmin -u root password 'test1234' ①
03:35:01	0x1f02fd000	0x4202fd000	c ②
03:35:02	0x1f081e800	0x42081e800	sudo apt-get insta ③
03:35:02	0x1f0890800	0x420890800	sudo apt-key ④
03:35:02	0x1f08ff000	0x4208ff000	mkdir key ⑤
03:35:03	0x1f1279000	0x421279000	netstat -anp tcp ⑥

Figure 16: Snapshot of bash keyboard buffer monitoring.

Description	Value
System	Intel Xeon E5-2650
# of Sockets (Nodes)	2 (1 per socket)
# of Cores	8 per node
Memory Capacity	8 GB per node
Interconnect	6.4 GT/s QPI
OS version	Linux Kernel 3.6.0

Table 3: Dell PE R620 server using in our evaluation.

user types any word on their own shell prompt, all characters are stored in a bash keyboard buffer in the memory unencrypted.

In the Bash shell (v4.3), the bash keyboard buffer is represented by a data structure referred to as `mhead`, as shown in Figure 15. Similar to the bash shell credential object attack described earlier, a fingerprint is necessary to detect this data structure in memory. For the fingerprint of the bash keyboard buffer, we use three unique values as the fingerprint based on the `mhead` data structure. The character variable (Figure 15①) has a unique 8 bits value (e.g., either 0x7F when allocated or 0x54 when freed). In addition, the 16-bit variable (Figure 15②) is always a predefined magic number (0x5555) and the 32-bit variable (Figure 15③) is always 0x100 that refers to the size of the buffer.

Based on the fingerprint, after PIkit is installed, an attacker can search for the fingerprint to gather information from victim users’ shell prompt, including potentially password information since data in the keyboard buffer is unencrypted. To evaluate this attack, we use the system described in Table 3 and assume a SSH server where multiple users use bash shell prompts from remote connections. Multiple remote SSH connections are made on the victim node and for each shell, different prompt inputs are used to evaluate the bash keyboard buffer attack.

By scanning for the fingerprint on the victim address region, we were able to monitor the bash keyboard buffer of other users. Different examples are shown in Figure 16 – Figure 16① shows a user typing in their password while Figure 16③ show other commands being typed by another user. In comparison, Figure 16② shows a keyboard buffer for another user that does not contain any content. Thus, with the buffer monitoring with PIkit installed, the dynamic information from other users’ keyboard input can be leaked.

5.3 Shared Library Attack

The attack described earlier in this section required both heap spraying and fingerprint scanning to obtain privi-

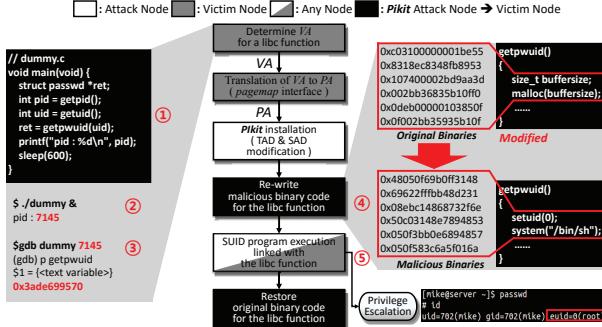


Figure 17: Shared library (libc) Attack with PIkit.

lege escalation. While this approach was successful, the attack can be time consuming because of the amount of time to scan the memory and attack reliability can become an issue since if the heap spraying does not fall within the victim memory region as the attack process needs to be repeated. To increase the attack reliability with PIkit, we take advantage of the target address decoder (TAD) structure available in the Intel Sandybridge architecture that provides the ability to “fix” the victim memory address region based on the attack memory address region. This approach is useful if the victim address region contains information that can be maliciously modified at a fixed physical memory location and does not change over time.

In this section, we take advantage of shared libraries that are often loaded into user memory space at single physical location and shared by different users. Once the shared library is initially loaded, the physical address of the library will likely not change. A commonly used shared library is `libc` and in this section, we exploit PIkit to overwrite existing functions in the `libc` with a malicious code to obtain privilege escalation. No fingerprint scanning or heap spraying is required and results in a highly reliable attack.

An overview of the `libc` attack is shown in Figure 17. To first obtain the virtual address of the `libc` shared library, an attacker can write a simple attack program (Figure 17①) which links `libc` dynamically. By executing a function within `libc` (e.g., `getpwuid()`) in the attack program, the `libc` library is dynamically linked. After obtaining the process ID for the attack program that loaded the shared library (Figure 17②), we use a debugger (e.g., `gdb`) on the running process to determine the virtual address of the `getpwuid()` function in the shared library (Figure 17③). The physical address corresponding to the virtual address of the `libc` library can be determined through the `/proc/pid/pagemap` interface and PIkit can be installed as described earlier in Section 4.5. Since the target victim address (i.e., `libc` function code location) is known, we modify the TAD structure offset accordingly, based on the attack address region obtained with PIkit.

After PIkit is installed, the attacker can re-write the runtime code loaded in the memory with user-level memory operation (Figure 17④). In our example attack, the first 48 bytes of `getpwuid()` function is re-written with malicious binary code that executes shell with root privilege. Even if the shared library is located in non-writable memory regions, PIkit bypasses any OS permission check ($W \oplus X^9$ based implementations [3, 52, 34]) and write the malicious code to the physical DRAM directly. The modified code executes `setuid(0)` to obtain root access but this does not work with user-level privilege and requires root access. However, the `passwd` Linux program has SUID (Set owner User ID) permission and is linked with `getpwuid()` function at runtime, the execution enables the malicious binary code to escalate the user privilege to the root (Figure 17⑤). Once privilege escalation is obtained, the attacker can restore the original binary of `getpwuid()` function to prevent the execution of the malicious code for other users that executes `getpwuid()`. While `getpwuid()` function for the `libc` was used to demonstrate this attack, other seldomly used library functions can be used or the malicious code can be modified to enable root shell only for specific user ID.

6 Discussion

In this section, we provide discussion on how PIkit can be potentially exploited for other types of attack, possible solutions both in software and hardware, as well as some limitations of PIkit.

6.1 VM Escape Attack

VM (virtual machine) escape is defined as enabling a VM to interact with the hypervisor directly and/or access other VMs running on the same host [30, 55]. If PIkit is installed on a multi-socket server that supports VMs, we expect that VM escape attack can be carried out. However, there are two challenges to implement VM escape with PIkit – huge pages and virtual address translation. PIkit exploited the availability of huge pages in modern OS to define the attack address region. Modern virtualization hardware technologies such as Intel VT-d support 1 GB huge pages [43] and hypervisors such as KVM, Xen and VMware also support 1GB huge page for guest VMs. We evaluated the VM escape attack possibility with Xen 4.4 but the hypervisor underneath the guest OS implemented the huge pages as a collection of smaller pages (e.g., 2 MB pages), likely because of implementation complexity. However, there is no fundamental reason why the hypervisor cannot support 1 GB

⁹A memory page must never be writable and executable at the same time.

Algorithm 1: PIkit monitor to detect modification to the DRAM address mapping table.

```

Input : monitoring
begin
  while monitoring do
    | - Get DRAM information from SPD
    | - Calculate Valid_address_ranges of installed
    |   DRAM from information
    |   A  $\leftarrow$  Valid_address_ranges
    | - Get Current_address_ranges from DRAM
    |   address mapping table
    |   B  $\leftarrow$  Current_address_ranges
    | if A  $\neq$  B then
    |   | PIkit detected
  return

```

huge pages.¹⁰ Another challenge is that address translation needs to be done twice to properly install PIkit – from the guest virtual address (gVA) to the guest physical address (gPA) within the VM and then, another translation to the machine physical address. While */proc/pid/pagemap* interface can be used for the translation from a gVA to a gPA for the VM local OS system, similar to the PIkit implementation described earlier, the hypervisor is responsible for another translation from gPA to machine PA. The hypervisor likely maintains a separate table/data structure for this translation and this needs to be reverse engineered to implement VM escape.

6.2 Possible Solutions

Possible solutions to PIkit can be classified as either a software-based or a hardware-based solution. The actual solution to PIkit is highly dependent upon the manufacturer of the hardware (e.g., Intel, AMD) as well as the system software used.

6.2.1 Exploit Existing Features

While evaluating PIkit on different systems, some of the recent AMD systems were not vulnerable to PIkit. To the best of our knowledge, the vulnerability was not removed for security reasons but removed for power saving implementation. C6 state is an ACPI defined CPU power saving state where the CPU is put in sleep mode and all CPU contexts are saved. In some AMD implementations, when the CPU enters the C6 state, the processor context is saved into a pre-defined region of the main memory. To avoid any possible corruption of the processor contexts that are saved, the AMD systems implement LockDramCfg option where some memory system related configurations cannot be modified, including the DRAM address mapping table [7]. However, the

¹⁰A very recent version of Xen (v 4.6) actually has support for 1 GB huge pages.

Description	Ratio (%)
SPD access period (I/O bound)	99.975 %
DRAM size calc period (CPU bound)	0.003 %
DRAM table access period (I/O bound)	0.019 %
Table Comparison period (CPU bound)	0.003 %

Table 4: Breakdown of CPU cycles ratio for the PIkit monitor. BIOS can disable the C6 state for some of these systems – which would enable PIkit to be installed. A simple solution for PIkit on such AMD systems is to always enable LockDramCfg to prevent PIkit from being installed. For the Intel systems, C6 power state is supported but the processor contexts are saved to the last level cache (and not the memory) to provide faster context switch – thus, to the best of our knowledge, similar LockDramCfg feature is not readily available in Intel x86 CPUs.

6.2.2 Software-based Solutions

Prior rootkit monitors can be extended to detect the presence of PIkit. The monitor continuously compares the value of the current DRAM address mapping table with the “correct” DRAM mapping table value – where the correct value is determined similar to how the DRAM mapping table is initialized by the firmware at boot time. Thus, if we assume the software monitor is protected with a secure platform [8, 9], the solution to detect PIkit can also be protected.

We implemented the PIkit monitor as a Linux kernel thread and we evaluate its performance overhead. High-level description of PIkit monitor is shown in Algorithm 1. We used the PARSEC 3.0 [12] and evaluated workloads with varying MPKI (misses per kilo-instructions), using the system described earlier in Table 2. To measure overall system performance, we run each workload with the number of threads equal to the number of physical cores in the system. Based on Linux kernel 3.6.0, we implemented the rootkit monitor as a loadable kernel module to avoid kernel code modification and re-compilation. The execution time with PIkit monitor is normalized to the baseline without the monitor and the performance overhead from the software is negligible as there is less than 2% impact on overall performance (Figure 18).

The analysis of the PIkit monitor overhead is shown in Table 4. The two CPU computation periods (e.g., DRAM size calculation and comparison) take only 0.006% of the total PIkit monitor execution time. In comparison, the other two I/O bound periods (e.g., SPD and PCI address access) occupy 99% of the monitor execution time since these accesses have long latency – resulting in the kernel thread waiting on the I/O and mostly experience uninterruptible Sleep state. Thus, the PIkit monitor and software solution has minimal impact on performance.

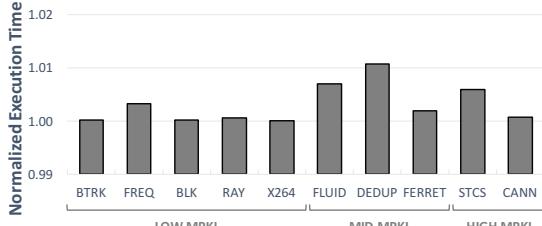


Figure 18: Performance overhead for PIkit monitor

6.2.3 Hardware-based Solutions

PIkit can be prevented with minimal hardware modifications. One hardware solution is to restrict the usage of the DRAM address mapping table entries used in multi-socket servers, based on the number of nodes in the system. If the hardware restricts the number of entries used to equal the number of nodes in the system, PIkit can be minimized. This approach does not completely remove the possibility of the PIkit since the attacker could use the entire local node’s memory as attack address region. However, unless the attacker is the only user on the local node, this can cause non-deterministic behavior. If the number of entries is restricted, the attacker can also modify the DRAM address mapping where only small region is specified as the attack address region but the remaining address range of the local node would not be mapped in the table. As a result, if any memory access occurs to an unmapped address region, the system will likely crash.

Another possible hardware solution is to design the DRAM address mapping table entries as write-once memory-mapped registers such that the DRAM mapping table can not be modified after it is initially written. A block diagram of such write-once register is shown in Figure 19 – after a write is done, the write enable (WE) to the register will be disabled and no further writes can be done to the registers unless system reset is asserted. This approach avoids any possibility of the PIkit attack since the DRAM address mapping table cannot be modified after it is initialized; however, this removes any flexibility in the system if CPU hotplug [38] or memory hotplug [27] is supported. If the system supports hotplug where the DRAM (or CPU) can be added or removed while the system is running, the DRAM address mapping table needs to be modified after it is initialized to reflect the change in the memory capacity. This would require a minor change to the hardware (i.e., OR’ing the RESET signal with another signal that detects a hotplug event). However, this can also open up other attack opportunities if the attacker has physical access to the memory modules – for example, doing a hotplug creates an opportunity to modify the DRAM address mapping table and install the PIkit.

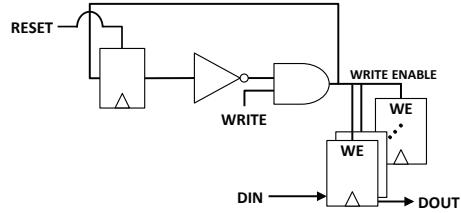


Figure 19: A PIkit hardware solution by creating a write-once register for the DRAM address mapping table.

6.3 Limitations

Memory mapping granularity: As described earlier in Section 4.2, there is a granularity (or the resolution) of the attack memory address region that can be specified with the DRAM address mapping table entries. In the AMD Opteron 6128 system, the smallest amount of memory address range that can be used as the attack address region is 16 MB. For the Intel Xeon E5-2650 (Sandybridge), the lower 26 bits are not specified and the granularity is 64 MBs. Thus, the granularity specifies the lower limit on the smallest attack address region that can be specified. This can be problematic if only normal page size is supported – i.e., with memory page size of 2 MBs, the attacker would require obtaining 8 *physical* pages to cover the 16 MBs attack region. If the entire attack region is not obtained by the attacker, other users (and programs) will access the victim address region and can cause unknown behaviors. However, huge pages that are available in modern OS can overcome this limitation.

DRAM coverage: One challenge of PIkit is the unknown DRAM physical mapping when an attack address is sent to the victim node. The details of how the DRAM physical address is mapped to the DRAM (i.e., row, column, channel, etc.) is vendor-specific. Although these details are often described in the specifications, the PIkit results in an unintended address arriving at the victim node memory system and it is not clear how the physical address bits are interpreted by the DRAM. For example, in the AMD Opteron 6128 system, the memory controller (or MCT) consists of two DRAM controllers (DCT0 and DCT1) where each DCT is responsible for half of the DRAM main memory connected to that particular node. When the attack node ID was *smaller* than the ID of the victim node (i.e., node 1 attacking node 2), only DCT0 address range could be accessed. However, by attacking node 2 from node 3, we discovered that DCT1 range can be accessed. Another limitation was that for some memory accesses, multiple attack addresses can map to the *same* victim address but this is a fundamental limitation of our proposed PIkit on the AMD system. In comparison, for the Intel Xeon E5-2650 (Sandybridge) server, the DRAM coverage issue was significantly minimized as there was an 1:1 memory mapping between the attack and the victim memory address region through the TAD structure. However, it is

not clear if this can be generalized to other Intel-based servers.

7 Conclusion

In this work, we described a new type of rootkit where the vulnerable hardware feature enables malicious activities to be carried out with only user-level code or payload. In particular, we presented PIkit – a processor-interconnect rootkit that enables an attacker to modify a packet’s destination and access victim’s memory region without proper permission. We described the design and challenges in implementing PIkit across both AMD and Intel x86 multi-socket servers. Once PIkit is installed, user-level codes used for malicious activities become very difficult to detect since memory accesses within the attack code appears as “normal” memory accesses to user-allocated memory. We demonstrated different malicious activities with PIkit, including bash shell credential object attack, keyboard buffer attack, and shared library attack.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. This research was supported in part by the Mid-career Researcher Program (NRF-2013R1A2A2A01069132), in part by IITP grant funded by MSIP (No.10041313, UX-oriented Mobile SW Platform), in part by the MSIP under the ITRC support program (IITP-2015-H8501-15-1005), and in part by Next-Generation Information Computing Development Program through NRF funded by the Ministry of Science, ICT & Future Planning (2015M3C4A7065647) and (No. NRF-2014M3C4A7030648).

References

- [1] chkrootkit. <http://www.chkrootkit.org/>.
- [2] Rootkit Hunter. https://rootkit.nl/projects/rootkit_hunter.html.
- [3] OpenBSD 3.3 release notes. <http://www.openbsd.org/33.html>, May 2003.
- [4] JEDEC Standard, SPD General Standard, 2008.
- [5] Intel® Xeon® Processor 7500 Series.
- [6] Linux Kernel Vulnerabilities Over Time. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33, 2015. [Online; accessed 19-Aug-2015].
- [7] ADVANCED MICRO DEVICES. BIOS and Kernel Developer Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, 2012.
- [8] ALVES, T., AND FELTON, D. TrustZone: Integrated Hardware and Software Security. *ARM white paper 3*, 4 (2004), 18–24.
- [9] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), vol. 13.
- [10] ANDERSON, D., AND TRODDEN, J. *Hypertransport System Architecture*. Addison-Wesley Professional, 2003.
- [11] AZAB, A. M., NING, P., AND ZHANG, X. SICE: A Hardware-Level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 375–388.
- [12] BIENIA, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [13] BIOS, A. kernel developers guide for amd family 10h processors, 2008.
- [14] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 27–38.
- [15] BUTLER, J. Direct Kernel Object Manipulation (DKOM). *Black Hat USA* (2004).
- [16] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (2011), ACM, p. 5.
- [17] CREED. Information about the Knark Rootkit. <http://ossec-docs.readthedocs.org/en/latest/rootcheck/rootcheck-knark.html>, 1999.
- [18] CUI, A., COSTELLO, M., AND STOLFO, S. J. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *NDSS* (2013).
- [19] DALLY, W. J., AND TOWLES, B. P. *Principles and Practices of Interconnection Networks*. Elsevier, 2004.
- [20] DAVID, F. M., CHAN, E. M., CARLYLE, J. C., AND CAMPBELL, R. H. Cloaker: Hardware Supported Rootkit Concealment. In *2008 IEEE Symposium on Security and Privacy (S&P’08)* (2008), IEEE, pp. 296–310.
- [21] HEASMAN, J. Implementing and Detecting an ACPI BIOS Rootkit. *Black Hat Federal 368* (2006).
- [22] HEATH, T., MARTIN, R. P., AND NGUYEN, T. D. Improving Cluster Availability Using Workstation Validation. In *ACM SIGMETRICS Performance Evaluation Review* (2002), vol. 30, ACM, pp. 217–227.
- [23] HOCK, R. Dica rootkit. <https://packetstormsecurity.com/files/26243/dica.tgz.html>, 2002.
- [24] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring Operating System Kernel Integrity with OSck. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 279–290.
- [25] HUND, R., HOLZ, T., AND FREILING, F. C. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *USENIX Security Symposium* (2009), pp. 383–398.
- [26] INTEL. Intel® Xeon® Processor E7-8800/4800/2800 v2 Product Family.
- [27] ISHIMATSU, Y. Memory Hotplug. *LinuxCon Japan* (2013).
- [28] KING, S. T., AND CHEN, P. M. SubVirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P’06)* (2006), IEEE, pp. 14–pp.
- [29] KLEEN, A. Virtual Memory Map with 4 level page tables. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2004.
- [30] KORTCHINSKY, K. CLOUDBURST: A VMware Guest to Host Escape Story. *Black Hat USA* (2009).

- [31] LEE, H., MOON, H., JANG, D., KIM, K., LEE, J., PAEK, Y., AND KANG, B. B. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *USENIX Security* (2013), pp. 511–526.
- [32] LOVE, R. *Linux Kernel Development*. Pearson Education, 2010.
- [33] MASKIEWICZ, J., ELLIS, B., MOURADIAN, J., AND SHACHAM, H. Mouse Trap: Exploiting Firmware Updates in USB Peripherals. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)* (2014).
- [34] MICROSOFT. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2. <http://support.microsoft.com/kb/875352>, 2008.
- [35] MILLER, T. Analysis of the T0rn rootkit. *SANS Institute* (2000).
- [36] MORGAN, T. P. X86 Servers Dominate The Datacenter-For Now. <http://www.nextplatform.com/2015/06/04/x86-servers-dominate-the-datacenter-for-now/>, 2015.
- [37] MUTNURY, B., PAGLIA, F., MOBLEY, J., SINGH, G. K., AND BELLOMIO, R. QuickPath Interconnect (QPI) Design and Analysis in High Speed Servers. In *19th Topical Meeting on Electrical Performance of Electronic Packaging and Systems* (2010), IEEE, pp. 265–268.
- [38] MWAIKAMBO, Z., RAJ, A., RUSSELL, R., SCHOPP, J., AND VADDAGIRI, S. Linux Kernel Hotplug CPU Support. In *Linux Symposium* (2004), vol. 2.
- [39] NIU, S., MO, J., ZHANG, Z., AND LV, Z. Overview of Linux Vulnerabilities. In *2nd International Conference on Soft Computing in Information Communication Technology* (2014), Atlantis Press.
- [40] NYBERG, C. M. SAdoor - A non listening remote shell and execution server. <http://krutibrko.sk/school/dp/samples/SAdoor/sadoor.pdf>, 2002.
- [41] OP, F. The FU rootkit. <https://www.soldierx.com/tools/FU-Rootkit>, 2008.
- [42] PETRONI JR, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium* (2004), San Diego, USA, pp. 179–194.
- [43] RIGHINI, M. Enabling Intel® Virtualization Technology Features and Benefits. *Intel White Paper*. Retrieved January 15 (2010), 2012.
- [44] RUSHANAN, M., AND CHECKOWAY, S. Run-DMA. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (2015).
- [45] RUTKOWSKA, J. Subverting the Vista Kernel For Fun And Profit. *SyScan* (2006).
- [46] SCHUSTER, A. Searching for Processes and Threads in Microsoft Windows Memory Dumps. *digital investigation 3* (2006), 10–16.
- [47] SHIELDS, T. Survey of Rootkit Technologies and Their Impact on Digital Forensics. http://www.donkeyonawaffle.org/misc/txs-rootkits_and_digital_forensics.pdf, 2008.
- [48] SOMER, L. Linux Rootkit 5. <https://packetstormsecurity.com/files/10533/lrk5.src.tar.gz.html>, 2000.
- [49] SONG, W., KIM, J., LEE, J.-W., AND ABTS, D. Security Vulnerability in Processor-Interconnect Router Design. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 358–368.
- [50] SPARKS, S., AND BUTLER, J. Shadow Walker: Raising The Bar For Windows Rootkit Detection. *Black Hat Japan 11*, 63 (2005), 504–533.
- [51] SPARKS, S., EMBLETON, S., AND ZOU, C. C. A Chipset Level Network Backdoor: Bypassing Host-Based Firewall & IDS. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (2009), ACM, pp. 125–134.
- [52] TEAM, P. Documentation for the PaX project - overall description. <http://pax.grsecurity.net/docs/pax.txt>, 2008.
- [53] VASISHT, V. R., AND LEE, H.-H. S. SHARK: Architectural Support for Autonomic Protection Against Stealth by Rootkit Exploits. In *2008 41st IEEE/ACM International Symposium on Microarchitecture* (2008), IEEE, pp. 106–116.
- [54] WANG, G., ESTRADA, Z. J., PHAM, C., KALBARTZYK, Z., AND IYER, R. K. Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring. In *9th USENIX workshop on offensive technologies (WOOT 15)* (2015).
- [55] WOJTCZUK, R., RUTKOWSKA, J., AND TERESHKIN, A. Xen Owning Trilogy. *Invisible Things Lab* (2008).
- [56] WOJTCZUK, R., AND TERESHKIN, A. Attacking Intel BIOS. *BlackHat, Las Vegas, USA* (2009).
- [57] ZADDACH, J., KURMUS, A., BALZAROTTI, D., BLASS, E.-O., FRANCILLON, A., GOODSPEED, T., GUPTA, M., AND KOLTSIDAS, I. Implementation and Implications of a Stealth Hard-Drive Backdoor. In *Proceedings of the 29th annual computer security applications conference* (2013), ACM, pp. 279–288.

Verifying Constant-Time Implementations

José Bacelar Almeida

HASLab - INESC TEC & Univ. Minho

Gilles Barthe

IMDEA Software Institute

Manuel Barbosa

HASLab - INESC TEC & DCC FCUP

François Dupressoir

IMDEA Software Institute

Michael Emmi

Bell Labs, Nokia

Abstract

The constant-time programming discipline is an effective countermeasure against timing attacks, which can lead to complete breaks of otherwise secure systems. However, adhering to constant-time programming is hard on its own, and extremely hard under additional efficiency and legacy constraints. This makes automated verification of constant-time code an essential component for building secure software.

We propose a novel approach for verifying constant-time security of real-world code. Our approach is able to validate implementations that locally and intentionally violate the constant-time policy, when such violations are benign and leak no more information than the public outputs of the computation. Such implementations, which are used in cryptographic libraries to obtain important speedups or to comply with legacy APIs, would be declared insecure by all prior solutions.

We implement our approach in a publicly available, cross-platform, and fully automated prototype, *ct-verif*, that leverages the *SMACK* and *Boogie* tools and verifies optimized *LLVM* implementations. We present verification results obtained over a wide range of constant-time components from the *NaCl*, *OpenSSL*, *FourQ* and other off-the-shelf libraries. The diversity and scale of our examples, as well as the fact that we deal with top-level APIs rather than being limited to low-level leaf functions, distinguishes *ct-verif* from prior tools.

Our approach is based on a simple reduction of constant-time security of a program P to safety of a product program Q that simulates two executions of P . We formalize and verify the reduction for a core high-level language using the *Coq* proof assistant.

1 Introduction

Timing attacks pose a serious threat to otherwise secure software systems. Such attacks can be mounted by measuring the execution time of an implementation directly

in the execution platform [23] or by interacting remotely with the implementation through a network. Notable examples of the latter include Brumley and Boneh’s key recovery attacks against *OpenSSL*’s implementation of the RSA decryption operation [15]; and the Canvel et al. [16] and Lucky 13 [4] timing-based padding-oracle attacks, that recover application data from SSL/TLS connections [38]. A different class of timing attacks exploit side-effects of cache-collisions; here the attacker infers memory-access patterns of the target program — which may depend on secret data — from the memory latency correlation created by cache sharing between processes hosted on the same machine [11, 31]. It has been demonstrated in practice that these attacks allow the recovery of secret key material, such as complete AES keys [21].

As a countermeasure, many security practitioners mitigate vulnerability by adopting so-called *constant-time* programming disciplines. A common principle of such disciplines governs programs’ control-flow paths in order to protect against attacks based on measuring execution time and branch-prediction attacks, requiring that paths do not depend on program secrets. On its own, this characterization is roughly equivalent to security in the program counter model [29] in which program counter values do not depend on program secrets. Stronger constant-time policies also govern programs’ memory-access patterns in order to protect against cache-timing attacks, requiring that accessed memory addresses do not depend on program secrets. Further refinements govern the operands of program operations, e.g., requiring that inputs to certain operations do not depend on program secrets, as the execution time of some machine instructions, notably integer division and floating point operations, may depend on the values of their operands.

Although constant-time security policies are the most effective and widely-used software-based countermeasures against timing attacks [11, 25, 20], writing constant-time implementations can be difficult. Indeed, doing so requires the use of low-level programming languages or

compiler knowledge, and forces developers to deviate from conventional programming practices. For instance, the program `if b then x := v1 else x := v2` may be replaced with the less conventional `x := b * v1 + (1 - b) * v2`. Furthermore, the observable properties of a program execution are generally not evident from its source code, e.g., due to optimizations made by compilers or due to platform-specific behaviours.

This raises the question of how to validate constant-time implementations. A recently disclosed timing leak in OpenSSL’s DSA signing [19] procedure demonstrates that writing constant-time code is complex and requires some form of validation. The recent case of Amazon’s s2n library also demonstrates that the deployment of less rigid timing countermeasures is extremely hard to validate: soon after its release, two patches¹ were issued for protection against timing attacks [3, 5], the second of which exploits a timing-related vulnerability introduced when fixing the first. These vulnerabilities eluded both extensive code review and testing, suggesting that standard software validation processes are an inadequate defense against timing vulnerabilities, and that more rigorous analysis techniques are necessary.

In this work, we develop a unifying formal foundation for constant-time programming policies, along with a formal and fully automated verification technique. Our formalism is parameterized by a flexible leakage model that captures the various constant-time policies used in practice, including path-based, address-based, and operand-based characterizations, wherein program paths, accessed memory addresses, and operand sizes, respectively, are independent of program secrets. Importantly, our formalism is precise with respect to the characterization of program secrets, distinguishing not only between public and private input values, but also between private and publicly observable output values. While this distinction poses technical and theoretical challenges, constant-time implementations in cryptographic libraries like OpenSSL include optimizations for which paths, addresses, and operands are contingent not only on public input values, but also on publicly observable output values. Considering only input values as non-secret information would thus incorrectly characterize those implementations as non-constant-time.

We demonstrate the practicality of our verification technique by developing a prototype, `ct-verif`, and evaluating it on a comprehensive set of case studies collected from various off-the-shelf libraries such as OpenSSL [25], NaCl [13], FourQlib [17] and curve25519-donna.² These examples include a diverse set of constant-time algorithms for fixed-point arithmetic, elliptic curve operations, and symmetric and public-key cryptography. Apart from in-

dicating which inputs and outputs should be considered public, the verification of our examples does not require user intervention, can handle existing (complete and non-modified) implementations, and is fully automated.

One strength of our verification technique is that it is agnostic as to the representation of programs and could be performed on source code, intermediate representations, or machine code. From a theoretical point of view, our approach to verifying constant-time policies is a sound and complete reduction of the security of a program P to the assertion-safety of a program Q , meaning that P is constant-time (w.r.t. the chosen policy) if and only if Q is assertion-safe. We formalize and verify the method for a core high-level language using the Coq proof assistant. Our reduction is inspired from prior work on self-composition [10, 37] and product programs [40, 9], and constructs Q as a product of P with itself—each execution of Q encodes two executions of P . However, our approach is unique in that it exploits the key feature of constant-time policies: program paths must be independent of program secrets. This allows a succinct construction for Q since each path of Q need only correspond to a single control path³ of P — path divergence of the two executions of P would violate constant-time. Our method is practical precisely because of this optimization: the product program Q has only as many paths as P itself, and its verification can be fully automated.

Making use of this reduction in practice raises the issue of choosing the programming language over which verification is carried out. On the one hand, to obtain a faithful correspondence with the executable program under attacker scrutiny, one wants to be as close as possible to the machine-executed assembly code. On the other hand, building robust and sustainable tools is made easier by existing robust and sustainable frameworks and infrastructure. Our `ct-verif` prototype performs verification of constant-time properties at the level of optimized LLVM assembly code, which represents a sweet spot in the design space outlined by the above requirements.

Indeed, performing verification after most optimization passes ensures that the program, which may have been written in a higher-level such as C, preserves the constant-time policy even after compiler optimizations. Further, stepping back from machine-specific assembly code to LLVM assembly essentially supports generic reasoning over *all* machine architectures—with the obvious caveat that the leakage model adopted at the LLVM level captures the leakage considered in all the practical lower-level languages and adversary models under study. This is a reasonable assumption, given the small abstraction gap between the two languages. (We further discuss the issues that may arise between LLVM and lower-level assembly

¹See pull requests #147 and #179 at github.com/awslabs/s2n.

²<https://code.google.com/p/curve25519-donna/>

³This is more subtle for programs with publicly observable outputs; see Section 4.

code when describing our prototype implementation.) Finally, our prototype and case studies justify that existing tools for LLVM are in fact sufficient for our purposes. They may also help inform the development of architecture-specific verification tools.

In summary, this work makes the following fundamental contributions, each described above:

- i. a unifying formal foundation for constant-time programming policies used in practice,
- ii. a sound and complete reduction-based approach to verifying constant-time programming policies, verified in Coq, and
- iii. a publicly available, cross-platform, and fully automated prototype implementing this technique on LLVM code, ct-verif, based on SMACK,
- iv. extensive case studies demonstrating the practical effectiveness of our approach on LLVM code, and supporting discussions on the wider applicability of the technique.

We begin in Section 2 by surveying constant-time programming policies. Then in Section 3 we develop a notion of constant-time security parameterized over leakage models, and in Section 4 we describe our reduction from constant-time security to assertion safety on product programs. Section 5 describes our implementation of a verifier for constant-time leveraging this reduction, and in Section 6 we study the verification of actual cryptographic implementations using our method. We discuss related work in Section 7, and conclude in Section 8.

2 Constant-Time Implementations

We now explain the different flavors of constant-time security policies and programming disciplines that enforce them, using small examples of problematic code that arise repeatedly in cryptographic implementations. Consider first the C function of Figure 1, that copies a sub-array of length `sub_len`, starting at index `l_idx`, from array `in` to array `out`. Here, `len` represents the length of array `in`.

```

1 void copy_subarray(uint8 *out, const uint8 *in,
2                    uint32 len, uint32 l_idx, uint32 sub_len) {
3     uint32 i, j;
4     for(i=0; j<0; i<len; i++) {
5         if (i >= l_idx) && (i < l_idx + sub_len) {
6             out[j] = in[i]; j++;
7         }
8     }

```

Figure 1: Sub-array copy: `l_idx` is leaked by PC.

Suppose now that the starting addresses and lengths of both arrays are public. What we mean by this is that, the user/caller of this function is willing to accept a contract expressed over the calling interface, whereby the starting addresses and lengths of both arrays may be leaked to an

attacker, whereas the value of the `l_idx` variable and the array contents must not. Then, although the overall execution time of this function may seem roughly constant because the loop is executed a number of times that can be inferred from a public input, it might still leak sensitive information via the control flow. Indeed, due to the `if` condition in line 4, an attacker that is able to obtain a program-counter trace would be able to infer the value of `l_idx`. This could open the way to timing attacks based on execution time measurements, such as the Canvel et al. [16] and Lucky 13 [4] attacks, or to branch-prediction attacks in which a co-located spy process measures the latency introduced by the branch-prediction unit in order to infer the control flow of the target program [1]. An alternative implementation that fixes this problem is shown in Figure 2.

```

1 uint32 ct_lt(uint32 a, uint32 b) {
2     uint32 c = a ^ ((a ^ b) | ((a - b) ^ b));
3     return (0 - (c >> (sizeof(c) * 8 - 1)));
4 }
5
6 void cp_copy_subarray(uint8 *out, const uint8 *in,
7                       uint32 len, uint32 l_idx, uint32 sub_len) {
8     uint32 i, j, in_range;
9     for(i=0; i<sub_len; i++) out[i]=0;
10    for(i=0, j=0; i<len; i++) {
11        in_range = 0;
12        in_range |= ~ct_lt(i, l_idx);
13        in_range &= ct_lt(i, l_idx+sub_len);
14        out[j] |= in[i] & in_range;
15        j = j + (in_range % 2);
16    }

```

Figure 2: Sub-array copy: constant control flow but `l_idx` is leaked by memory access address trace.

Observe that the control flow of this function is now totally independent of `l_idx`, which means that it is constant for fixed values of all public parameters. However, this implementation allows a different type of leakage that could reveal `l_idx` to a stronger class of timing adversaries. Indeed, the memory accesses in line 13 would allow an attacker with access to the full trace of memory addresses accessed by the program to infer the value of `l_idx`—note that the sequence of `j` values will repeat at 0 until `l_idx` is reached, and then increase. This leakage could be exploited via cache-timing attacks [11, 31], in which an attacker controlling a spy process co-located with this program (and hence sharing the same cache) would measure its own memory access times and try to infer sensitive data leaked to accessed addresses from cache hit/miss patterns.

Finally, the program above also includes an additional potential leakage source in line 14. Here, the value of `j` is updated as a result of a DIV operation whose execution time, in some processors,⁴ may vary depending on the

⁴This is a quotation from the Intel 64 and IA-32 architectures ref-

values of its operands. This line of code might therefore allow an attacker that can take fine-grained measurements of the execution time to infer the value of `l_idx` [25]. There are two possible solutions for this problem: either ensure that the ranges of operands passed to such instructions are consistently within the same range, or use different algorithms or instructions (potentially less efficient) whose execution time does not depend on their operands. We note that, for this class of timing attackers, identifying leakage sources and defining programming disciplines that guarantee adequate mitigation becomes highly platform-specific.

An implementation of the same function that eliminates all the leakage sources we have identified above—assuming that the used native operations have operand-independent execution times—is given in Figure 3.

```

1 uint32 ct_eq(uint32 a, uint32 b) {
2     uint32 c = a ^ b;
3     uint32 d = ~c & (c - 1);
4     return (0 - (d >> (sizeof(d) * 8 - 1)));
5 }
6
7 void ct_copy_subarray(uint8 *out, const uint8 *in,
8     uint32 len, uint32 l_idx, uint32 sub_len) {
9     uint32 i, j;
10    for(i=0;i<sub_len;i++) out[i]=0;
11    for(i=0;i<len;i++) {
12        for(j=0;j<sub_len;j++) {
13            out[j] |= in[i] & ct_eq(l_idx+j, i);
14        }
15    }

```

Figure 3: Constant-time sub-array copy.

It is clear that the trade-off here is one between efficiency and security and, indeed, constant-time implementations often bring with them a performance penalty. It is therefore important to allow for relaxations of the constant-time programming disciplines when these are guaranteed *not* to compromise security. The example of Figure 4, taken from the NaCl cryptographic library [13] illustrates an important class of optimizations that arises from allowing leakage which is known to be *benign*.

This code corresponds to a common sequence of operations in secure communications: first verify that an incoming ciphertext is authentic (line 11) and, if so, recover the enclosed message (line 12) cleaning up some spurious data afterwards (line 13). The typical contract drawn at the function’s interface states that the secret inputs to the function include only the contents of the secret

erence manual: *The throughput of “DIV/IDIV r32” varies with the number of significant digits in the input EDX:EAX and/or of the quotient of the division for a given size of significant bits in the divisor r32. The throughput decreases (increasing numerical value in cycles) with increasing number of significant bits in the input EDX:EAX or the output quotient. The latency of “DIV/IDIV r32” also varies with the significant bits of the input values. For a given set of input values, the latency is about the same as the throughput in cycles.*

```

1 int crypto_secretbox_open(unsigned char *m,
2     const unsigned char *c, unsigned long long clen,
3     const unsigned char *n,
4     const unsigned char *k)
5 {
6     int i;
7     unsigned char subkey[32];
8     if (clen < 32) return -1;
9     crypto_stream_salsa20(subkey, 32, n, k);
10    if (crypto_auth_hmacsha512_verify(c, c+32, clen
11        -32, subkey) != 0) return -1;
12    crypto_stream_salsa20_xor(m, c, clen, n, k);
13    for (i = 0; i < 32; ++i) m[i] = 0;
14    return 0;
15 }

```

Figure 4: Verify-then-decrypt: verification result is publicly observable and can be leaked by control-flow.

key array. Now suppose we ensure that the functions called by this code are constant-time. Even so, this function is *not* constant-time: the result of the verification in line 11 obviously depends on the secret key value, and it is used for a conditional return statement.

The goal of this return statement is to reduce the execution time by preventing a useless decryption operation when the ciphertext is found to be invalid. Indeed, an authenticated decryption failure is typically publicly signaled by cryptographic protocols, in which case this blatant violation of the constant-time security policy would actually *not* constitute an additional security risk. Put differently, the potentially sensitive bit of information revealed by the conditional return is actually benign leakage: it is safe to leak it because it will be revealed anyway when the return value of the function is later made public. Such optimization opportunities arise whenever the target application accepts a contract at the function interface that is enriched with information about publicly observable outputs, and this information is sufficient to classify the extra leakage as benign.

The above examples motivate the remainder of the work in this paper. It is clear that checking the correct enforcement of constant-time policies is difficult. Indeed, the programming styles that need to be adopted are very particular to this domain, and degrade the readability of the code. Furthermore, these are non-functional properties that standard software development processes are not prepared to address. These facts are usually a source of criticism towards constant-time implementations. However, our results show that such criticism is largely unjustified. Indeed, our verification framework stands as proof that the strictness of constant-time policies makes them suitable for automatic verification. This is not the case for more lenient policies that are less intrusive but offer less protection (e.g., guaranteeing that the total execution time varies within a very small interval, or that the same number of calls is guaranteed to be made to a hash compression function).

In the next section we formalize constant-time security following the intuition above, as well as the foundations for a new formal verification tool that is able to automatically verify their correct enforcement over real-world cryptographic code.

3 A Formalization of Constant-Time

In order to reason about the security of the code actually executed after compilation, we develop our constant-time theory and verification approach on a generic unstructured assembly language, in Appendix A. In the present section we mirror that development on a simple high-level structured programming language for presentational clarity. We consider the language of *while programs*, enriched with arrays and assert/assume statements. Its syntax is listed in Figure 5. The metavariables x and e range over program variables and expressions, respectively. We leave the syntax of expressions unspecified, though assume they are deterministic, side-effect free, and that array expressions are non-nested.

$$\begin{aligned} p ::= & \text{skip} \mid x[e_1] := e_2 \mid \text{assert } e \mid \text{assume } e \mid p_1; p_2 \\ & \mid \text{if } e \text{ then } p_1 \text{ else } p_2 \mid \text{while } e \text{ do } p \end{aligned}$$

Figure 5: The syntax of while programs.

Although this language is quite simple, it is sufficient to fully illustrate our theory and verification technique. We include arrays rather than scalar program variables to model constant-time policies which govern indexed memory accesses. We include the assert and assume statements to simplify our reduction from the security of a given program to the assertion-safety of another. Figure 6 lists the semantics of while programs, which is standard.

$$\begin{array}{c} s' = s[\langle x, s(e_1) \rangle \mapsto s(e_2)] \quad s' = s \text{ if } s(e) \text{ else } \perp \\ \frac{}{\langle s, x[e_1] := e_2 \rangle \rightarrow \langle s', \text{skip} \rangle} \quad \frac{}{\langle s, \text{assert } e \rangle \rightarrow \langle s', \text{skip} \rangle} \\ \frac{s(e) = \text{true}}{\langle s, \text{assume } e \rangle \rightarrow \langle s, \text{skip} \rangle} \quad \frac{\langle s, p_1 \rangle \rightarrow \langle s', p'_1 \rangle}{\langle s, p_1; p_2 \rangle \rightarrow \langle s', p'_1; p_2 \rangle} \\ \frac{}{\langle s, \text{skip}; p \rangle \rightarrow \langle s, p \rangle} \quad \frac{}{\langle s, \text{if } e \text{ then } p_1 \text{ else } p_2 \rangle \rightarrow \langle s, p_i \rangle} \\ \frac{p' = (p; \text{while } e \text{ do } p) \text{ if } s(e) \text{ else skip}}{\langle s, \text{while } e \text{ do } p \rangle \rightarrow \langle s, p' \rangle} \end{array}$$

Figure 6: The operational semantics of while programs. All rules are guarded implicitly by the predicate $s \neq \perp$, and we abbreviate the predicate $s(e) = \text{true}$ by $s(e)$.

A *state* s maps variables x and indices $i \in \mathbb{N}$ to values $s(x, i)$, and we write $s(e)$ to denote the value of expression e in state s . The distinguished *error state* \perp represents

a state from which no transition is enabled. A *configuration* $c = \langle s, p \rangle$ is a state s along with a program p to be executed, and an *execution* is a sequence $c_1 c_2 \dots c_n$ of configurations such that $c_i \rightarrow c_{i+1}$ for $0 < i < n$. The execution is *safe* unless $c_n = \langle \perp, _ \rangle$; it is *complete* if $c_n = \langle _, \text{skip} \rangle$; and it is an *execution of program p* if $c_1 = \langle _, p \rangle$. A program p is *safe* if all of its executions are safe.

A *leakage model* L maps program configurations c to *observations* $L(c)$, and extends to executions, mapping $c_1 c_2 \dots c_n$ to the *observation* $L(c_1 c_2 \dots c_n) = L(c_1) \cdot L(c_2) \dots L(c_n)$, where ϵ is the identity observation, and $L(c) \cdot \epsilon = \epsilon \cdot L(c) = L(c)$. Two executions α and β are *indistinguishable* when $L(\alpha) = L(\beta)$.

Example 1. The baseline path-based characterization of constant-time is captured by leakage models which expose the valuations of branch conditions:

$$\begin{aligned} \langle s, \text{if } e \text{ then } p_1 \text{ else } p_2 \rangle &\mapsto s(e) \\ \langle s, \text{while } e \text{ do } p \rangle &\mapsto s(e) \end{aligned}$$

In this work we assume that all leakage models include the mappings above.

Example 2. Notions of constant-time which further include memory access patterns are captured by leakage models which expose addresses accessed in load and store instructions. In our simple language of while programs, this amounts to exposing the indexes to program variables read and written at each statement. For instance, the assignment statement exposes indexes read and written (the base variables need not be leaked as they can be inferred from the control flow):

$$\langle s, x_0[e_0] := e \rangle \mapsto s(e_0) s(e_1) \dots s(e_n)$$

where $x_1[e_1], \dots, x_n[e_n]$ are the indexed variable reads in expression e (if any exist).

Example 3. Notions of constant-time which are sensitive to the size of instruction operands, e.g., the operands of division instructions, are captured by leakage models which expose the relevant leakage:

$$\langle s, x[e_1] := e_2 / e_3 \rangle \mapsto S(e_2, e_3)$$

where S is some function over the operands of the division operation, e.g., the maximum size of the two operands.

In Section 2 we have intuitively described the notion of a contract drawn at a function’s interface; the constant-time security policies are defined relatively to this contract, which somehow defines the acceptable level of (benign) leakage that can be tolerated. Formally, we capture these contracts using a notion of equivalence between initial states (for a set X_i of inputs declared to be public) and

final states (for a set X_o of outputs declared to be publicly observable), as follows.

Given a set X of program variables, two configurations $\langle s_1, _\rangle$ and $\langle s_2, _\rangle$ are *X-equivalent* when $s_1(x, i) = s_2(x, i)$ for all $x \in X$ and $i \in \mathbb{N}$. Executions $c_1 \dots c_n$ and $c'_1 \dots c'_{n'}$ are *initially X-equivalent* when c_1 and c'_1 are *X-equivalent*, and *finally X-equivalent* when c_n and $c'_{n'}$ are *X-equivalent*.

Definition 1 (Constant-Time Security). A program is secure when all of its initially X_i -equivalent and finally X_o -equivalent executions are indistinguishable.

Intuitively, constant-time security means that any two executions whose input and output values differ only with respect to secret information must leak exactly the same observations. Contrasting our definition with other information flow policies, we observe that constant-time security asks that every two complete executions starting with X_i -equivalent states and ending with X_o -equivalent final states must be indistinguishable, while termination-insensitive non-interference asks that every two complete executions starting with X_i -equivalent states must end with X_o -equivalent final states. This makes the constant-time policies we consider distinct from the baseline notions of non-interference studied in language-based security. However, our policies can be understood as a specialized form of delimited release [34], whereby *escape hatches* are used to specify an upper bound on the information that is allowed to be declassified. Our notion of security is indeed a restriction of delimited release where escape hatches—our public output annotations—may occur only in the final state.

4 Reducing Security to Safety

The construction of the output-insensitive product of a program (with itself) is shown in Figure 7. It begins by assuming the equality of each public input $x \in X_i$ with its renamed copy \hat{x} , then recursively applies a guard and instrumentation to each subprogram. Guards assert the equality of leakage functions for each subprogram p and its variable-renaming \hat{p} .

```

product(p)    assume x=ŷ for x ∈ X_i;
               together(p)
together(p)   guard(p);
               instrument[λ p.(p; ũ), together](p)
guard(p)      assert L(p)=L(ũ)

```

Figure 7: Output-insensitive product construction.

Instrumentation preserves the control structure of the original program. Our construction uses the program instrumentation given in Figure 8, which is parameterized by functions α and β transforming assignments and

subprograms, respectively. In our constructions, α is either the identity function or else duplicates assignments over renamed variables, and β applies instrumentation recursively with various additional logics.

	instrument[α, β]($_\right)$
—	
skip	skip
$x[e_1] := e_2$	$\alpha(x[e_1] := e_2)$
assert e	assert e
assume e	assume e
$p_1; p_2$	$\beta(p_1); \beta(p_2)$
if e then p_1 else p_2	if e then $\beta(p_1)$ else $\beta(p_2)$
while e do p	while e do $\beta(p)$

Figure 8: Instrumentation for product construction.

Our first result states that this construction provides a reduction from constant-time security to safety that is sound for all safe input programs (i.e., a security verdict is always correct) and complete for programs where information about public outputs is not taken into consideration in the security analysis (i.e., an insecurity verdict is always correct).

Theorem 1. A safe program with (respectively, without) public outputs is secure if (respectively, iff) its output-insensitive product is safe.

Proof. First, note that program semantics is deterministic, i.e., for any two complete executions of a program p from the same state s_0 to states s_1 and s_2 emitting observations $L(\vec{c}_1)$ and $L(\vec{c}_2)$, respectively, we have $s_1 = s_2$ and $L(\vec{c}_1) = L(\vec{c}_2)$. The product construction dictates that an execution of $\text{product}(p)$ from state $s \uplus \hat{s}$ reaches state $s' \uplus \hat{s}'$ if and only if the two corresponding executions of p leak the same observation sequence, from s to s' and from \hat{s} to \hat{s}' , where \hat{s} is the variable-renaming of s . \square

In order to deal with program paths which depend on public outputs, we modify the product construction, as shown in Figure 9, to record the observations along output-dependent paths in history variables and assert their equality when paths merge. The output-sensitive product begins and ends by assuming the equality of public inputs and outputs, respectively, with their renamed copies, and finally asserts that the observations made across both simulated executions are identical. Besides delaying the assertion of observational indistinguishability until the end of execution, when outputs are known to be publicly observable, this construction allows paths to diverge at branches which depend on these outputs, checking whether both executions alone leak the same recorded observations.

Technically, this construction therefore relies on identifying the branches, i.e., the if and while statements, whose conditions can only be declared benign when public outputs are considered. This has two key implications.

```

product( $p$ )   same_observations := true;
               assume  $x = \hat{x}$  for  $x \in X_i$ ;
               together( $p$ );
               assume  $x = \hat{x}$  for  $x \in X_o$ ;
               assert same_observations

together( $p$ ) if benign( $p$ ) then
                $h := \varepsilon$ ;  $\hat{h} := \varepsilon$ ;
               alone $_h(p)$ ;
               alone $_{\hat{h}}(\hat{p})$ ;
               same_observations &&:=  $h = \hat{h}$ 
            otherwise
               guard( $p$ );
               instrument[ $\lambda p. (p, \hat{p})$ , together]( $p$ )
guard( $p$ )   same_observations &&:=  $L(p) = L(\hat{p})$ 
alone $_h(p)$  record $_h(p)$ ;
               instrument[ $\lambda p. p$ , alone $_h$ ]( $p$ )
record $_h(p)$   $h += L(p)$ 

```

Figure 9: Output-sensitive product construction

First, it either requires a programmer to annotate which branches are benign in a public-sensitive sense, or additional automation in the verifier, e.g., to search over the space of possible annotations; in practice the burden appears quite low since very few branches will need to be annotated as being benign. Second, it requires the verifier to consider separate paths for the two simulated executions, rather than a single synchronized path. While this deteriorates to an expensive full product construction in the worst case, in practice these output-dependent branches are localized to small non-nested regions, and thus asymptotically insignificant.

Theorem 2. *A safe program is secure iff its output-sensitive product is safe with some benign-leakage annotation.*

Proof. Completeness follows from completeness of self-composition, so only soundness is interesting. Soundness follows from the fact that we record history in the variables h and \hat{h} whenever we do not assert the equality of observations on both sides. \square

Coq formalization The formal framework presented in this and the previous section has been formalized in Coq. Our formalization currently includes the output-insensitive reduction from constant-time security to safety of the product program as described in Figures 7 and 8, for the while language in Figure 5. We prove the soundness and completeness of this reduction (Theorem 1) following the intuition described in the sketch presented above. Formalization of the output-sensitive construction and the proof of Theorem 2 should not present any additional difficulty, other than a more intricate case analysis when control flow may diverge. Our Coq formalization serves two purposes: i. it rigorously captures the theoretical foundations of our approach and complements the intu-

itive description we gave above; and ii. it could serve as a template for a future formalization of the machine-level version of these same results, which underlies the implementation of our prototype and is presented in Appendices A and B. A Coq formalization of this low-level transformation could be integrated with CompCert, providing more formal guarantees on the final compiled code.

5 Implementation of a Security Verifier

Using the reduction of Section 4 we have implemented a prototype, ct-verif, which is capable of automatically verifying the compiled and optimized LLVM code resulting from compiling actual C-code implementations of several constant-time algorithms. Before discussing the verification of these codes in Section 6, here we describe our implementation and outline key issues. Our implementation and case studies are publicly available⁵ and cross-platform. ct-verif leverages the SMACK verification tool [32] to compile the annotated C source via Clang⁶ and to optimize the generated assembly code via LLVM⁷ before translating to Boogie⁸ code. We perform our reduction on the Boogie code, and apply the Boogie verifier (which performs verification using an underlying SMT⁹ logic solver) to the resulting program.

5.1 Security Annotations

We provide a simple annotation interface via the following C function declarations:

```

void public_in(smack_value_t);
void public_out(smack_value_t);
void benign_branching();

```

where `smack_value_t` values are handles to program values obtained according to the following interface

```

smack_value_t __SMACK_value();
smack_value_t __SMACK_values(void* ary,
                           unsigned count);
smack_value_t __SMACK_return_value(void);

```

and `__SMACK_value(x)` returns a handle to the value stored in program variable `x`, `__SMACK_values(ary, n)` returns a handle to an `n`-length array `ary`, and `__SMACK_return_value()` provides a handle to the procedure's return value. While our current interface does not provide handles to entire structures, non-recursive structures can still be annotated by annotating the handles to each of their (nested) fields. Figure 10 demonstrates the annotation of a decryption function for the Tiny Encryption Algorithm (TEA). The first argument `v` is a pointer to

⁵<https://github.com/imdea-software/verifying-constant-time>

⁶C language family frontend for LLVM: <http://clang.llvm.org>

⁷The LLVM Compiler Infrastructure: <http://llvm.org>

⁸Boogie: <http://github.com/boogie-org/boogie>

⁹Satisfiability Modulo Theories: <http://smtlib.cs.uiowa.edu>

a public ciphertext block of two 32-bit words, while the second argument `k` is a pointer to a secret key.

```

1 void decrypt_cpa_wrapper(uint32_t* v, uint32_t* k) {
2     public_in(__SMACK_value(v));
3     public_in(__SMACK_value(k));
4     public_in(__SMACK_values(v, 2));
5     decrypt(v, k);
6 }
```

Figure 10: Annotations for the TEA decryption function.

5.2 Reasoning about Memory Separation

In some cases, verification relies on establishing separation of memory objects. For instance, if the first of two adjacent objects in memory is annotated as public input, while the second is not, then a program whose branch conditions rely on memory accesses from the first object is only secure if we know that those accesses stay within the bounds of the first object. Otherwise, if those accesses might occur within the second object, then the program is insecure since the branch conditions may rely on private information.

Luckily SMACK has builtin support for reasoning about the separation of memory objects, internally leveraging an LLVM-level data-structure analysis [26] (DSA) to partition memory objects into disjoint regions. Accordingly, the generated Boogie code encodes memory as several disjoint map-type global variables rather than a single monolithic map-type global variable, which facilitates scalable verification. This usually provides sufficient separation for verifying security as well. In a few cases, DSA may lack sufficient precision. In those settings, it would be possible to annotate the source code with additional assumptions using SMACK’s `__Verifier_assume()` function. This limitation is not fundamental to our approach, but instead an artifact of design choices¹⁰ and code rot¹¹ in DSA itself.

5.3 Product Construction for Boogie Code

The Boogie intermediate verification language (IVL) is a simple imperative language with well-defined, clean, and mathematically-focused semantics which is a convenient representation for performing our reduction. Conceptually there is little difference between performing our shadow product reduction at the Boogie level as opposed to the LLVM or machine-code level since the Boogie code produced by SMACK corresponds closely to the LLVM code, which is itself similar to machine code. Indeed our machine model of Appendix A is representative. Practically however, Boogie’s minimal syntax greatly facilitates

¹⁰DSA is designed to be extremely scalable at the expense of precision, yet such extreme scalability is not necessary for our use.

¹¹See the discussion thread at <https://groups.google.com/forum/?#!topic/llvm-dev/pnU5ecuvr6c>.

our code-to-code translation. In particular, shadowing the machine state amounts to making duplicate copies of program variables. Since memory accesses are represented by accesses to map-type global variables, accessing a shadowed address space amounts to accessing the duplicate of a given map-type variable.

Our prototype models observations as in Examples 1 and 2 of Section 3, exposing the addresses used in memory accesses and the values used as branch conditions as observations. According to our construction of Section 4, we thus prefix each memory access by an assertion that the address and its shadow are equal, and prefix each branch by an assertion that the condition and its shadow are equal. Finally, for procedures with annotations, our prototype inserts assume statements on the equality of public inputs with their shadows at entry blocks.

When dealing with public outputs, we perform the output-sensitive product construction described in Section 4 adapted to an unstructured assembly language. Intuitively, our prototype delays assertions (simply by keeping track of their conjunction in a special variable) but otherwise produces the standard output-insensitive product program. It then replaces the blocks corresponding to the potentially desynchronized conditional with blocks corresponding to the output-sensitive product construction that mixes control and data product. Finally, it inserts code that saves the current assertions before the region where the control flow may diverge, and restores them afterwards, making sure to also take into account the assertions collected in between.

5.4 Scalability of the Boogie Verifier

Since secure implementations, and cryptographic primitives in particular, do not typically call recursive procedures, we instruct Boogie to inline all procedures during verification. This avoids the need for manually written procedure contracts, or for sophisticated procedure specification inference tools.

Program loops are handled by automatically computing loop invariants. This is fairly direct in our setting, since invariants are simply conjunctions of equalities between some program variables and their shadowed copies. We compute the relevant set of variables by taking the intersection of variables live at loop heads with those on which assertions inserted by our reduction depend.

5.5 Discussion

ct-verif is based on a theoretically sound and complete methodology; however, practical interpretations of its results must be analyzed with care. First, leakage models are constructed, and in our case are based on LLVM rather than machine code. Second, verification tools can

be incomplete, notably because of approximations made by sub-tasks performed during verification (for instance, data-structure analysis or invariant inference).

Therefore, it is important to evaluate ct-verif empirically, both on positive and negative examples. Our positive experimental results in the next section demonstrate that the class of constant-time programs that is validated automatically by ct-verif is significantly larger than those tackled by existing techniques and tools. Our negative examples, available from the public repository,¹² are taken from known buggy libraries (capturing the recent CacheBleed attack,¹³ in particular), and others taken to illustrate particularly tricky patterns. Again, some of these examples illustrate the value of a low-level verification tool by exhibiting compilation-related behaviours. Unsurprisingly, we found that there is little value in running our tool on code that was *not* written to be constant-time. Conversely, we found that our tool can be helpful in detecting subtle breaches in code that was written with constant-time in mind, but was still insecure, either due to subtle programming errors, or to compilation-related issues.

It remains to discuss possible sources of unsoundness that may arise from our choice of LLVM as the target for verification (rather than actual machine code). As highlighted in Section 1, this choice brings us many advantages, but it implies that our prototype does not strictly know what machine instructions will be activated and on which arguments, when the final code is actually executed. For example, our assumptions on the timing of a particular LLVM operation may not hold for the actual processor instruction that is selected to implement this operation in executable code. Nevertheless we argue that the LLVM assembly code produced just before code generation sufficiently similar to *any* target-machine’s assembly code to provide a high level of confidence. Indeed, the majority of compiler optimizations are made prior to code generation. At the point of code generation, the key difference between representations is that in LLVM assembly:

- i. some instruction/operand types may not be available on a given target machine,
- ii. there are arbitrarily-many registers, whereas any given machine would have a limited number, and
- iii. the order of instructions within basic blocks is only partially determined.

First we note that neither of these differences affects programs’ control-flow paths, and the basic-block structure of programs during code generation is generally preserved. Second, while register allocation does generally change memory-access patterns, spilled memory accesses are generally limited to the addresses of scalar stack variables, which are fully determined by control-flow paths. Thus

both path-based and address-based constant-time properties are generally preserved. Operand-based constant-time properties, however, are generally not preserved: it is quite possible that instruction selection changes the types of some instruction’s operands, implying a gap between LLVM and machine assembly regarding whether operand sizes may depend on secrets. Dealing with such sources of leakage requires architecture-specific modeling and tools, which are out of the scope of a research prototype.

6 Experimental Results

We evaluate ct-verif on a large set of examples, mostly taken from off-the-shelf cryptographic libraries, including the pervasively used OpenSSL [25] library, the NaCl [13] library, the FourQlib library [17], and curve25519-donna.¹⁴ The variety and number of cryptographic examples we have considered is unprecedented in the literature. Furthermore, our examples serve to demonstrate that ct-verif outperforms previous solutions in terms of scale (the sizes of some of our examples are orders of magnitude larger than what could be handled before), coverage (we can handle top-level public APIs, rather than low-level or leaf functions) and robustness (ct-verif is based on a technique which is not only sound, but also complete).

All execution times reported in this section were obtained on a 2.7GHz Intel i7 with 16GB of RAM. Size statistics measure the size in lines of code (loc) of the analyzed Boogie code (similar in size to the analyzed LLVM bitcode) before inlining. When presenting time measurements, all in seconds, we separate the time taken to produce the product program (annotating it with the \times symbol) from that taken to verify it: in particular, given a library, the product program can be constructed once and for all before verifying each of its entry points. ct-verif assumes that the leakage trace produced by standard library functions `memcpy` and `memset` depends only on their arguments (that is, the address and length of the objects they work on, rather than their contents). This is a mild assumption that can be easily checked for each platform. For examples that use dynamic memory allocation, such as the OpenSSL implementation of PKCS#1 padding, ct-verif enforces that `malloc` and `free` are called with secret-independent parameters and assumes that the result of `malloc` is always secret-independent in this case. In other words, we assume that the address returned by `malloc` depends only on the trace of calls to `malloc` and `free`, or that the memory allocator observes only the memory layout to make allocation decisions.¹⁵

¹²<https://github.com/imdea-software/verifying-constant-time>

¹³<https://ssrg.nicta.com.au/projects/TS/cachebleed>

¹⁴<https://code.google.com/p/curve25519-donna/>

¹⁵It may be possible to extend this to an allocator that also has access to the trace of memory accesses, since they are made public.

Example	Size	Time (x)	Time
tea	200	2.33	0.47
rlwe_sample	400	5.78	0.65
nacl_salsa20	700	5.60	1.11
nacl_chacha20	10000	8.30	1.92
nacl_sha256_block	20000	27.7	4.17
nacl_sha512_block	20000	39.49	4.29

Table 1: Verification of crypto primitives.

6.1 Cryptographic Primitives

For our first set of examples, we consider a representative set of cryptographic primitives: a standard implementation of TEA [39] (tea), an implementation of sampling in a discrete Gaussian distribution by Bos et al. [14] (rlwe_sample) and several parts of the NaCl library [13] library.

Table 1 gives the details (we include only a subset of the NaCl verification results listed as nacl_xxxx). The verification result for rlwe_sample only excludes its core random byte generator, essentially proving that this core primitive is the only possible source of leakage. In particular, if its leakage and output bytes are independent, then the implementation of the sampling operation is constant-time. Verification of the SHA-256 implementation in NaCl above refers to the compression function; the full implementation of the hash function, which simply iterates this compression function, poses a challenge to our prototype due to the action of DSA: the internal state of the function is initialized as a single memory block, that later stores both secret and public values that are accessed separately. This issue was discussed in Section 5.2, where we outlined a solution using assume statements.

6.2 TLS Record Layer

To further illustrate scalability to large code bases, we now consider problems related to the MAC-then-Encode-then-CBC-Encrypt (MEE-CBC) construction used in the TLS record layer to obtain an authenticated encryption scheme. This construction is well-understood from the perspective of provable security [24, 30], but implementations have been the source of several practical attacks on TLS via timing side-channels [16, 4].

We apply our prototype to two C implementations of the MEE-CBC decryption procedure, treating only the input ciphertext as public information. Table 2 shows the corresponding verification results. We extract the first implementation from the OpenSSL sources (version 0.9.8zg). It includes all the countermeasures against timing attacks currently implemented in the MEE-CBC component in OpenSSL as documented in [25]. We verify the parts of the code that handle MEE-CBC decryption (1K loc of C, or 10K loc in Boogie): i. decryption of the encrypted message using AES128 in CBC

Example	Time (x)	Time
mee-cbc-openssl	10.6	18.73
mee-cbc-nacl	24.64	92.56

Table 2: Verification of MEE-CBC TLS record layer.

mode; ii. removing the padding and checking its well-formedness; iii. computing the HMAC of the unpadded message, even for bad padding, and using the same number of calls to the underlying hash compression function (in this case SHA-1); and iv. comparing the transmitted MAC to the computed MAC in constant-time. Our verification does not include the SHA1 compression function and AES-128 encryption—these are implemented in assembly—and hence our result proves that the only possible leakage comes from these leaf functions. (In OpenSSL the SHA1 implementation is constant-time but AES-128 makes secret-dependent memory accesses.)

As our second example, we consider a full 800 loc (in C, 20K loc in Boogie) implementation of MEE-CBC [5], which includes the implementation of the low level primitives (taken from the NaCl library).

Our prototype is able to verify the constant-time property of both implementations—with only the initial ciphertext and memory layout marked as public. Perhaps surprisingly, our simple heuristic for loop invariants is sufficient to handle complex control-flow structures such as the one shown in Figure 11, taken from OpenSSL.

```

1 k = 0;
2 if /* low cond */ { k = /* low exp */ }
3 if (k > 0)
4 { for (i = 1; i < k / /* low var */; i++)
5   { /* i-dependent memory access */ }
6 }
7 for (i = /* low var */; i <= /* low var */; i++)
8 { /* i-dependent memory access */
9   for (j = 0; j < /* low var */; j++)
10   { if (k < /* low var */)
11     /* k-dependent memory access */
12   else if (k < /* low exp */)
13     /* k-dependent memory access */
14     k++;
15   }
16   for (j = 0; j < /* low var */; j++)
17   { /* j-dependent memory access */ }
18 }
```

Figure 11: Complex control-flow from OpenSSL.

6.3 Fixed-Point Arithmetic

Our third set of examples is taken from the libfixedtimefixedpoint library, developed by Andryesco et al. [7] to mitigate several attacks due to operand-dependent leakage in the timing of floating point operations. In the conclusion of the paper we discuss how our prototype can be extended to deal with the vulnerable code that was attacked in [7]. Here we present

Function	Size	Time
fix_eq	100	1.45
fix_cmp	500	1.44
fix_mul	2300	1.50
fix_div	1000	1.53
fix_ln	11500	2.66
fix_convert_from_int64	100	1.43
fix_sin	800	1.64
fix_exp	2200	1.62
fix_sqrt	1400	1.55
fix_pow	18000	1.42 ¹⁶

Table 3: Verification of `libfixedtimefixedpoint`.

our verification results over the library that provides an alternative secure constant-time solution

The `libfixedtimefixedpoint` library (ca. 4K loc of C or 40K loc in Boogie) implements a large number of fixed-point arithmetic operations, from comparisons and equality tests to exponentials and trigonometric functions. Core primitives are automatically generated parametrically in the size of the integer part: we verify code generated with the default parameters. As far as we know, this is the first application of verification to this floating point library.

Table 3 shows verification statistics for part of the library. We verify all arithmetic functions without any inputs marked as public, but display only some interesting data points here. We discuss the `fix_pow` function, during whose execution the code shown in Figure 12 is executed on a `frac` array that is initialized as a “0” string literal. The function in which this snippet appears is not generally constant-time, but it is always used in contexts where all indices of `frac` that are visited up to and including the particular index that might trigger a sudden loop exit at line 6 contains public (or constant) data. Thanks to our semantic characterization of constant-time policies, ct-verif successfully identifies that the leakage produced by this code is indeed benign, whereas existing type-based or taint-propagation based would mark this program as insecure.

```

1 uint64_t result = 0;
2 uint64_t extra = 0;
3
4 for(int i = 0; i < 20; i++) {
5     uint8_t digit = (frac[i] - (uint8_t)'0');
6     if (frac[i] == '\0') { break; }
7     result += ((uint64_t)digit) * pow10[i];
8     extra += ((uint64_t)digit) * pow10_extra[i];
9 }
```

Figure 12: `fix_pow` code.

¹⁶We manually provide an invariant of the form $\exists i_{\max}. 0 \leq i < i_{\max} \leq 20 \wedge \text{frac}[i_{\max}] == 0 \wedge \forall j. 0 \leq j \leq i_{\max} \Rightarrow \text{public}(\text{frac}[j])$ for the loop shown in Figure 12. Loop unrolling could also be used, since the loop is statically bounded.

Example	Size	Time (x)	Time
curve25519-donna	10000	10.18	456.97
FourQLib	-	7.87	-
eccmadd	2500	-	133.72
eccdouble	3000	-	70.67
eccnorm	3500	-	156.48
point_setup	600	-	0.99
R1_to_R2	2500	-	7.92
R5_to_R1	2000	-	1.26
R1_to_R3	2500	-	2.42
R2_to_R4	1000	-	0.93

Table 4: Verification of elliptic curve arithmetic.

6.4 Elliptic Curve Arithmetic

As a final illustrative example of the capabilities of ct-verif in handling existing source code from different sources, we consider two constant-time implementations of elliptic curve arithmetic: the `curve25519-donna` implementation by Langley,¹⁷ and the `FourQLib` library [17]. The former library provides functions for computing essential elliptic curve operations over the increasingly popular Curve25519 initially proposed by Bernstein [12], whereas the latter uses a recently proposed alternative high-performance curve. Table 4 shows the results.

For `curve25519-donna`, we verify the functional entry point, used to generate public points and shared secrets, assuming only that the initial memory layout is public.

For `FourQLib`, we verify all the core functions for point addition, doubling and normalization, as well as coordinate conversions, all under the only assumption that the addresses of the function parameters are public. ct-verif successfully detects expected dependencies of the execution time on public inputs in the point validation function `ecc_point_validate`.

6.5 Publicly Observable Outputs

We wrap up this experimental section by illustrating the flexibility of the output-sensitive product construction, and how it permits expanding the coverage of real-world crypto implementations in comparison with previous approaches. As a first example we consider an optimized version of the `mee-cbc-nacl` example. Instead of using a constant-time select and zeroing loop to return its result (as shown in Figure 13, where the return code `res` is secret-dependent and marked as public and `in_len` is a public input), the code branches on the return code as shown in Figure 14. (The rest of the code is unmodified, and therefore constant-time.)

This is similar to the motivating example that we presented in Section 2, but here the goal is to avoid the unnecessary cleanup loop at the end of the function in executions where it is not needed. Again, because the return code is made public when it is returned to the caller,

¹⁷<https://code.google.com/p/curve25519-donna/>

```

1 good = ~((res == RC_SUCCESS) - 1);
2 for(i = 0; i < in_len; i++) { out[i] &= good; }
3 *out_len &= good;

```

Figure 13: MEE-CBC decryption: constant-time.

```

1 if (res != RC_SUCCESS) {
2     for(i = 0; i < in_len; i++) { out[i] = 0; }
3     *out_len = 0;
4 }

```

Figure 14: MEE-CBC decryption: constant-time.

this control-flow dependency on secret information can be classified as benign leakage. The output-sensitive product constructed by our prototype for this example, when the displayed conditional is annotated as benign leakage, verifies in slightly less than 2 minutes. The additional computation cost of verifying this version of the program may be acceptable when compared to the performance gains in the program itself—however minor: verification costs are one off, whereas performance issues in the cryptographic library are paid per execution.

```

1 int RSA_padding_check_PKCS1_type_2(uchar *to, int
2 tlen, const uchar *from, int flen, int num)
3 {
4     int i, zero_index = 0, msg_index, mlen = -1;
5     uchar *em = NULL;
6     uint good, found_zero_byte;
7
8     if (tlen < 0 || flen < 0) return -1;
9     if (flen > num) goto err;
10    if (num < 11) goto err;
11
12    em = OPENSSL_zalloc(num);
13    if (em == NULL) return -1;
14    memcpy(em + num - flen, from, flen);
15
16    good = ct_is_zero(em[0]);
17    good &= ct_eq(em[1], 2);
18
19    found_zero_byte = 0;
20    for (i = 2; i < num; i++) {
21        uint equals0 = ct_is_zero(em[i]);
22        zero_index = ct_select_int(~found_zero_byte &
23        equals0, i, zero_index);
24        found_zero_byte |= equals0;
25
26        good &= ct_ge((uint)(zero_index), 2 + 8);
27        msg_index = zero_index + 1;
28        mlen = num - msg_index;
29        good &= ct_ge((uint)(tlen), (uint)(mlen));
30
31        /* We can't continue in constant-time because we
32         * need to copy the result and we cannot fake
33         * its length. This unavoidably leaks timing
34         * information at the API boundary. */
35        if (!good) { mlen = -1; goto err; }
36        memcpy(to, em + msg_index, mlen);
37
38    err:
39        OPENSSL_free(em);
40        return mlen;
41    }

```

Figure 15: RSA PKCS1 padding check from OpenSSL

Finally, we present in Figure 15 an RSA PKCS1.5

padding check routine extracted from OpenSSL (similar code exists in other cryptographic libraries, such as boringssl¹⁸). The developers note the most interesting feature of this code in the comment on line 30: although this function is written in the constant-time style, the higher-level application (here referred to as an API boundary) does not give this implementation enough information to continue without branching on data dependent from secret inputs (here, the contents of `from`). One way in which this could be achieved would be for the function to accept an additional argument indicating some public bound on the expected message length. The constant-time techniques described previously in this paper could then be used to ensure that the leakage depends only on this additional public parameter. However, given the constraint forced upon the implementer by the existing API, the final statements in the function must be as they are, leading to (unavoidable and hence) benign leakage. Using our techniques, this choice can be justified by declaring the message length returned by the function as being (the only) public output that is safe to leak. Note that `flen`, `tlen` and `num` are public, and hence declaring `mlen` as a public output provides sufficient information to verify the control-flow leakage in line 31, and also the accessed addresses in line 32 as being benign. Verifying the output-sensitive product program when `mlen` is marked as a public output takes under a second.¹⁹

This example shows that dealing with relaxations of the constant-time policies enabled by output-sensitive API contracts is important when considering functions that are directly accessible by the adversary, rather than internal functions meant to be wrapped. Dealing with these use cases is an important asset of our approach, and is a problem not considered by previous solutions.

7 Related Work

Product programs Product constructions are a standard tool in the algorithmic analysis of systems. Product programs can be viewed as their syntactic counterpart. Systematic approaches for defining and building product programs are considered in [9]. Three instances of product programs are most relevant to our approach: self-composition [10] and selective self-composition [37], which have been used for proving information flow security of programs, and cross-products [40], which have been used for validating the correctness of compiler optimizations. We review the three constructions below, obliterating their original purpose, and presenting them from the perspective of our work.

¹⁸<https://boringssl.googlesource.com/>

¹⁹With simple implementations of `OPENSSL_zalloc` and `OPENSSL_free` that wrap standard memory functions.

The self-composition of a program P is a program Q which executes P twice, sequentially, over disjoint copies of the state of P . Self-composition is *sound and complete*, in the sense that the set of executions of program Q is in 1-1 bijection with the set of pairs of executions of program P . However, reasoning about self-composed programs may be very difficult, as one must be able to correlate the actions of two parts of program Q which correspond to the same point of execution of the original program P .

The cross-product Q of a program P coerces the two copies of P to execute in lockstep, by inserting assert statements at each branching instruction. Cross-product is not complete for all programs, because pairs of executions of program P whose control-flow diverge result in an unsafe execution of program Q . As a consequence, one must prove that Q is safe in order to transfer verification results from Q to P . However, cross-product has a major advantage over self-composition: reasoning about cross-products is generally easier, because the two parts of program Q which correspond to the same point of execution of the original program P are adjacent.

Selective self-composition is an approach which alternates between cross-product and self-composition, according to user-provided (or inferred for some applications) annotations. Selective self-composition retains the soundness and completeness of self-composition whilst achieving the practicality of cross-product.

Our output-insensitive product construction (Figure 7) is closely related to cross-product. In particular, Theorem 1 implies that cross-products characterize constant-path programs. We emphasize that, for this purpose, the incompleteness of cross-products is not a limitation but a desirable property. On the other hand, our output-sensitive product construction (Figure 9) is closely related to selective self-composition.

Language-based analysis/mitigation of side-channels
Figure 16 summarizes the main characteristics of several tools for verifying constant-time security, according to the level at which they carry out the analysis, the technique they use, their support for public inputs and outputs, their soundness and completeness, and their usability. ct-verif is the only one to support publicly observable outputs, and the only one to be sound, theoretically complete and practical. Moreover, we argue that extending any of these tools to publicly observable outputs is hard; in particular, several of these tools exploit the fact that cryptographic programs exhibit “abnormally straight line code behavior”, and publicly observable outputs are precisely used to relax this behavior. We elaborate on these points below.

FlowTracker [33] implements a precise, flow sensitive, constant-time (static) analysis for LLVM programs. This tool takes as input C or C++ programs with security annotations and returns a positive answer or a counterexample.

Tool	Target	Analysis method	Inputs/Outputs	Sound/Complete	Usability
tis-ct	C	static	X/X	✓/X	✓(a)
ABPV [6]	C	logical	✓/X	✓/✓	X(b)
VirtualCert	x86	static	✓/X	✓/X	X(c)
FlowTracker	LLVM	static	✓/X	✓/X	✓
ctgrind	binary	dynamic	✓/X	X/X	✓
CacheAudit	binary	static	X/X	✓/X	X(d)
This work	LLVM	logical	✓/✓	✓/✓	✓

Figure 16: Comparison of different tools. Target indicates the level at which the analysis is performed. Input/Outputs classifies whether the tool supports public inputs and publicly observable outputs. Usability includes coverage and automation. (a): requires manual interpretation of dependency analysis. (b): requires interactive proofs. (c): requires code rewriting. (d): supports restricted instruction set.

FlowTracker is incomplete (i.e. rejects secure programs), and it does not consider publicly observable outputs.

VirtualCert [8] instruments the CompCert certified compiler [27] with a formally verified, flow insensitive type system for constant-time security. It takes as input a C program with security annotations and compiles it to (an abstraction of) x86 assembly, on which the analysis is performed. VirtualCert imposes a number of restrictions on input programs (so off-the-shelf programs must often be adapted before analysis), is incomplete, and does not support publicly observable outputs.

ctgrind²⁰ is an extension of Valgrind that verifies constant-address security. It takes an input a program with taint annotations and returns a yes or no answer. ctgrind is neither sound nor complete and does not support publicly observable outputs.

tis-ct is an extension of the FramaC platform for analyzing dependencies in C programs and helping towards proving constant-time security.²¹ tis-ct has been used to analyze OpenSSL. Rather than a verification result, tis-ct outputs a list of all input regions that may flow into the leakage trace, as well as the code locations where that flow may occur. Although this does not directly allow the verification of adherence to a particular security policy, we note that checking that the result list is a subset of public inputs could provide, given an appropriate annotation language, a verification method for public input policies. Since it relies on a dependency analysis rather than semantic criteria, tis-ct is incomplete.

Almeida, Barbosa, Pinto and Vieira [6] propose a methodology based on deductive verification and self-composition for verifying constant-address security of C

²⁰<https://github.com/agl/ctgrind/>.

²¹<http://trust-in-soft.com/tis-ct/>

implementations. Their approach extends to constant-address security earlier work by Svenningsson and Sands [36] for constant-path security. This approach does not consider publicly observable outputs and it does not offer a comparable degree of automation to the one we demonstrate in this paper.

CacheAudit [18] is a static analyzer for quantifying cache leakage in a *single run* of a binary program. CacheAudit takes as input a binary program (in a limited subset of 32-bit x86, e.g. no dynamic jump) and a leakage model, but *no* security annotation (there is no notion of public or private, neither for input, nor output). CacheAudit is sound with respect to a simplified machine code semantics, rather than to a security policy. However, it is incomplete.

There are many other works that develop language-based methods for side-channel security (not necessarily in the computational model of this paper). Agat [2] proposes a type-based analysis for detecting timing leaks and a type-directed transformation for closing leaks in an important class of programs. Molnar, Piotrowski, Schultz and Wagner [29] define the program counter security model and a program transformation for making programs secure in this model. Other works include [28, 35, 41].

8 Conclusion

This paper leaves interesting directions for future work. We intend to improve ct-verif in two directions. First, we shall enhance enforcement of more expressive policies, such as those taking into consideration input-dependent instruction execution times. The work of Andryesco et al. [7] shows that variations in the timing of floating point processor operations may lead to serious vulnerabilities in non-cryptographic systems. Dealing with such timing leaks requires reasoning in depth about the semantics of a program, and is beyond the reach of techniques typically used for non-interference analysis. Our theoretical framework inherits this ability from self-composition, and this extension of ct-verif hinges solely on the effort required to embed platform-specific policy specifications into the program instrumentation logics of the prototype. As a second improvement to ct-verif, we will add support for SSE instructions, which are key to reconciling high-speed and security, for example in implementing AES [22].

Acknowledgements The first two authors were funded by Project “TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020”, which is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European

Regional Development Fund (ERDF). The third and fourth authors were supported by projects S2013/ICE-2731 N-GREENS Software-CM and ONR Grants N000141210914 (AutoCrypt) and N000141512750 (SynCrypt). The fourth author was also supported by FP7 Marie Cure Actions-COFUND 291803 (Amarout II).

We thank Peter Schwabe for providing us with a collection of negative examples. We thank Hovav Shacham, Craig Costello and Patrick Longa for helpful observations on our verification results.

References

- [1] Onur Aciicmez, Cetin Kaya Koc, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *2007 ACM Symposium on Information, Computer and Communications security (ASI-ACCS'07)*, pages 312–320. ACM Press, 2007.
- [2] Johan Agat. Transforming out Timing Leaks. In *Proceedings POPL'00*, pages 40–53. ACM, 2000.
- [3] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on Amazon’s s2n implementation of TLS. Cryptology ePrint Archive, Report 2015/1129, 2015. Available at <http://eprint.iacr.org/>. To appear in proceedings of EuroCrypt, 2016.
- [4] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy, SP 2013*, pages 526–540. IEEE Computer Society, 2013.
- [5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and Francois Dupressoir. Verifiable side-channel security of cryptographic implementations: constant-time mee-cbc. Cryptology ePrint Archive, Report 2015/1241, 2015. Available at <http://eprint.iacr.org/>. To appear in proceedings of Fast Software Encryption, 2016.
- [6] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.*, 78(7):796–812, 2013.
- [7] Marc Andryesco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 623–639. IEEE Computer Society, 2015.

- [8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 1267–1279. ACM Press, November 2014.
- [9] Gilles Barthe, Juan Manuel Crespo, and Cesar Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *Formal Methods*, volume 6664 of *LNCS*. Springer-Verlag, 2011.
- [10] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. Secure Information Flow by Self-Composition. In R. Focardi, editor, *Computer Security Foundations*, pages 100–114. IEEE Press, 2004.
- [11] Daniel J. Bernstein. Cache-timing attacks on aes, 2005. <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>.
- [12] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006.
- [13] Daniel J. Bernstein. Cryptography in NaCl, 2011. <http://nacl.cr.yp.to>.
- [14] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 553–570. IEEE Computer Society, 2015.
- [15] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.
- [16] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 583–599. Springer, Heidelberg, August 2003.
- [17] Craig Costello and Patrick Longa. FourQ: Four-dimensional decompositions on a Q-curve over the mersenne prime. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 214–235. Springer, 2015.
- [18] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Usenix Security 2013*, 2013.
- [19] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. “Make sure DSA signing exponentiations really are constant-time”. *Cryptology ePrint Archive*, Report 2016/594, 2016.
- [20] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. “make sure dsa signing exponentiations really are constant-time”. *Cryptology ePrint Archive*, Report 2016/594, 2016. <http://eprint.iacr.org/2016/594>.
- [21] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505. IEEE Computer Society, 2011.
- [22] Mike Hamburg. Accelerating AES with vector permute instructions. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 18–32, 2009.
- [23] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO’96*, volume 1109 of *LNCS*, pages 104–113. Springer, Heidelberg, August 1996.
- [24] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 310–331. Springer, Heidelberg, August 2001.
- [25] Adam Langley. Lucky thirteen attack on TLS CBC. Imperial Violet, February 2013. <https://www.imperialviolet.org/2013/02/04/luckythirteen.html>, Accessed October 25th, 2015.
- [26] Chris Lattner, Andrew Lenhardt, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 278–289. ACM, 2007.

- [27] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 42–54. ACM, 2006.
- [28] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *CSF 2013*, pages 51–65, 2013.
- [29] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *ICISC 05*, volume 3935 of *LNCS*, pages 156–168. Springer, Heidelberg, December 2006.
- [30] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389. Springer, Heidelberg, December 2011.
- [31] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.
- [32] Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014.
- [33] Bruno Rodrigues, Fernando Pereira, and Diego Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of Compiler Construction*, 2016. To appear.
- [34] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, pages 174–191, 2003.
- [35] Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 718–735. Springer, 2013.
- [36] Josef Svenningsson and David Sands. Specification and verification of side channel declassification. In *FAST'09*, volume 5983 of *LNCS*, pages 111–125. Springer, 2009.
- [37] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *SAS'2005*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
- [38] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 534–546. Springer, Heidelberg, April / May 2002.
- [39] David J. Wheeler and Roger M. Needham. TEA, a tiny encryption algorithm. In Bart Preneel, editor, *FSE'94*, volume 1008 of *LNCS*, pages 363–366. Springer, Heidelberg, December 1995.
- [40] Anna Zaks and Amir Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2008.
- [41] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 503–516, 2015.

A Machine-Level Constant-Time Security

In this section we formalize constant-time security policies at the instruction set architecture (ISA) level. Modeling and verification at this low-level captures security of the actual executable code targeted by attacker scrutiny, which can differ significantly from the original source code prior to compilation and optimization. In order to describe our verification approach in a generic setting, independently of the computing platform and the nature of the exploits, we introduce abstract notions of *machines* and *observations*.

A.1 An Abstract Computing Platform

Formally, a *machine* $M = \langle A, B, V, I, O, F, T \rangle$ consists of

- a set A of address-space names,
- a set B of control-block names, determining the set $K = \{\text{fail}, \text{halt}, \text{spin}, \text{next}, \text{jump}(b) : b \in B\}$ of control codes,
- a set V of values — each address space $a \in A$ corresponds to a subset $V_a \subset V$ of values; together, the address spaces and values determine the set $S = A \rightarrow (V_a \rightarrow V)$ of states, each $s \in S$ mapping $a \in A$ to value store $s_a : V_a \rightarrow V$; we write $a:v$ to denote a reference to $s_a(v)$,
- a set I of instructions — operands are references $a:v$, block names $b \in B$, and literal values,
- a set O of observations, including the null observation ϵ ,
- a leakage function $F : S \times I \rightarrow O$ determining the observation at each state-instruction pair, and
- a transition function $T : S \times I \rightarrow S \times K$ from states and instructions to states and control codes.

We assume that the value set V includes the integer value $0 \in \mathbb{N}$, that the control-block names include `entry`, and that the instruction set I includes the following:

$$T(s, \text{assume } a:v) = \begin{cases} \langle s, \text{spin} \rangle & \text{if } s_a(v) = 0 \\ \langle s, \text{next} \rangle & \text{otherwise} \end{cases}$$

$$T(s, \text{assert } a:v) = \begin{cases} \langle s, \text{fail} \rangle & \text{if } s_a(v) = 0 \\ \langle s, \text{next} \rangle & \text{otherwise} \end{cases}$$

$$T(s, \text{goto } b) = \langle s, \text{jump}(b) \rangle$$

$$T(s, \text{halt}) = \langle s, \text{halt} \rangle$$

We write $s[a:v_1 \mapsto v_2]$ to denote the state s' identical to s except that $s'_a(v_1) = v_2$.

Programs are essentially blocks containing instructions. Formally, a *location* $\ell = \langle b, n \rangle$ is a block name $b \in B$ and index $n \in \mathbb{N}$; the location $\langle \text{entry}, 0 \rangle$ is called the *entry location*, and L denotes the set of all locations. The location $\langle b, n \rangle$ is the *next successor* of $\langle b, n - 1 \rangle$ when $n > 0$, and is the *start of block* b when $n = 0$. A *program for machine M* is a function $P : L \rightarrow I$ labeling locations with instructions.

A.2 Semantics of Abstract Machines

A *configuration* $c = \langle s, \ell \rangle$ of machine M consists of a state $s \in S$ along with a location $\ell \in L$, and is called

- *initial* when ℓ is the entry location,

- *failing* when $T(s, P(\ell)) = \langle _, \text{fail} \rangle$,
- *halting* when $T(s, P(\ell)) = \langle _, \text{halt} \rangle$, and
- *spinning* when $T(s, P(\ell)) = \langle _, \text{spin} \rangle$.

The *observation* at c is $F(s, P(\ell))$, and configuration $\langle s_2, \ell_2 \rangle$ is the *successor* of $\langle s_1, \ell_1 \rangle$ when $T(s_1, P(\ell_1)) = \langle s_2, k \rangle$ and

- $k = \text{next}$ and ℓ_2 is the next successor of ℓ_1 ,
- $k = \text{jump}(b_2)$ and ℓ_2 is the start of block b_2 , or
- $k = \text{spin}$ and $\ell_2 = \ell_1$.

We write $c_1 \rightarrow c_2$ when c_2 is the successor of c_1 , and C denotes the set of configurations.

An *execution of program P for machine M* is a configuration sequence $e = c_0 c_1 \dots \in (C^* \cup C^\omega)$ such that $c_{i-1} \rightarrow c_i$ for each $0 < i < |e|$, and $c_{|e|-1}$ is failing or halting if $|e|$ is finite, in which case we say that e is failing or halting, respectively. The *trace* of e is the sequence $o_0 o_1 \dots$ of observations of at $c_0 c_1 \dots$ concatenated, where $o \cdot \epsilon = \epsilon \cdot o = o$. Executions with the same trace are *indistinguishable*.

Definition 2 (Safety). A program P on machine M is safe when no executions fail. Otherwise, P is unsafe.

A.3 Constant-Time Security

To define our security property we must relate program traces to input and output values, since, generally speaking, the observations made along executions should very well depend on, e.g., publicly-known input values. Security thus relies on distinguishing program inputs and outputs as public or private. We make this distinction formally using address spaces, supposing that machines include

- a public input address space $i \in A$, and
- a publicly observable output address space $o \in A$, in addition to, e.g., register and memory address spaces.

Intuitively, the observations made on a machine running a secure program should depend on the initial machine state only in the public input address space; when observations depend on non-public inputs, leakage occurs. More subtly, observations which are independent of public inputs can still be made, so long as each differing observation is eventually justified by differing publicly observable output values. Otherwise we consider that leakage occurs. Formally, we say that that two states $s_1, s_2 \in S$ are *a-equivalent* for $a \in A$ when $s_1(a)(v) = s_2(a)(v)$ for all $v \in V_a$. Executions e_1 from state s_1 and e_2 from state s_2 are *initially a-equivalent* when s_1 and s_2 are *a-equivalent*, and finite executions to s'_1 and s'_2 are *finally a-equivalent* when s'_1 and s'_2 are *a-equivalent*.

Definition 3 (Constant-Time Security). A program is secure when:

1. Initially i-equivalent and finally o-equivalent executions are indistinguishable.
2. Initially i-equivalent infinite executions are indistinguishable.

Otherwise, P is insecure.

The absence of publicly observable outputs simplifies this definition, since the executions of public-output-free programs are finally o-equivalent, trivially.

B Reducing Security to Safety

According to standard notions, security is a property over pairs of executions: a program is secure so long as executions with the same public inputs and public outputs are indistinguishable. In this section we demonstrate a reduction from security to safety, which is a property over single executions. The reduction works by instrumenting the original program with additional instructions which simulate two executions of the same program side-by-side, along the same control locations, over two separate address spaces: the original, along with a *shadow* of the machine state. In order for our reduction to be sound, i.e., to witness all security violations as safety violations, the control paths of the simulated executions must not diverge unless they yield distinct observations — in which case our reduction yields a safety violation. This soundness requirement can be stated via the following machine property, which amounts to saying that control paths can be leaked to an attacker.

Definition 4 (Control Leaking). A machine M is control leaking if for all states $s \in S$ and instructions $i_1, i_2 \in I$ the transitions $T(s, i_1) = \langle _, k_1 \rangle$ and $T(s, i_2) = \langle _, k_2 \rangle$ yield the same control codes $k_1 = k_2$ whenever the observations $F(s, i_1) = F(s, i_2)$ are identical.

For the remainder of this presentation, we suppose that machines are control leaking. This assumption coincides with that of Section 4: all considered leakage models expose the valuations of branch conditions. Besides control leaking, our construction also makes the following modest assumptions:

- address spaces can be separated and renamed, and
- observations are accessible via instructions.

We capture the first requirement by assuming that programs use a limited set $A_1 \subset A$ of the possible address-space names, and fixing a function $\alpha : A_1 \rightarrow A_2$ whose range $A_2 \subset A$ is disjoint from A_1 . We then lift this function from address-space names to instructions, i.e., $\alpha : I \rightarrow I$, by replacing each reference $a:v$ with $\alpha(a):v$. We capture the second requirement by assuming the existence of a function $\beta : I \times A \times V \rightarrow I$ such that

$$T(s, \beta(i, a, v)) = \langle s[a:v \mapsto F(s, i)], \text{next} \rangle.$$

For a given instruction $i \in I$, address space $a \in A$, and value $v \in V$, the instruction $\beta(i, a, v)$ stores the observation $F(s, i)$ in state $s \in S$ at $a:v$.

Following the development of Section 4, we develop an *output-insensitive* reduction which is always sound, but complete only for programs without publicly-annotated outputs. The extension to an *output-sensitive* reduction which is both sound and complete for all programs mirrors that developed in Section 4. This extension is a straightforward adaptation of Section 4’s from high-level structured programs to low-level unstructured programs, thus we omit it here.

Assume machines include a vector-equality instruction

$$T(s, \text{eq } a_x : \vec{x} \ a_y : \vec{y} \ a : z) = \langle s[a : z \mapsto v], \text{next} \rangle$$

where \vec{x} and \vec{y} are equal-length vectors of values, and $v = 0$ iff $s(a_x)(x_n) \neq s(a_y)(y_n)$ for some $0 \leq n < |\vec{x}|$. This requirement is for convenience only; technically only a simple scalar-equality instruction is necessary.

To facilitate the checking of initial/final range equivalences for security annotations we assume that a given program P has only a single `halt` instruction, and

$$\begin{aligned} P(\text{entry}, 0) &= \text{goto } b_0 \\ P(\text{exit}, 0) &= \text{halt}. \end{aligned}$$

This is without loss of generality since any program can easily be rewritten in this form. Given the above functions α and β , and a fresh address space a , the *shadow product* of a program P is the program P_\times defined by an entry block which spins unless the public input values in both i and $\alpha(i)$ address spaces are equal,

$$P_\times(\text{entry}, n) = \begin{cases} \text{eq } i : V_i \ \alpha(i) : V_i \ a : x & n = 0 \\ \text{assume } a : x & n = 1 \\ \text{goto } b_0 & n > 1 \end{cases}$$

an exit block identical to the original program,

$$P_\times(\text{exit}, n) = P(\text{exit}, n)$$

and finally a rewriting of every other block $b \notin \{\text{entry}, \text{exit}\}$ of P to run each instruction on two separate address spaces,

$$P_\times(b, n) = \begin{cases} \beta(i, a, x) & n = 0 \pmod{6} \\ \beta(\alpha(i), a, y) & n = 1 \pmod{6} \\ \text{eq } a : x \ a : y \ a : z & n = 2 \pmod{6} \\ \text{assert } a : z & n = 3 \pmod{6} \\ i & n = 4 \pmod{6} \\ \alpha(i) & n = 5 \pmod{6} \end{cases}$$

where $i = P(b, n/6)$

while asserting that the observations of each instruction are the same along both simulations.

Theorem 3. A safe program P with (respectively, without) public outputs is secure if (respectively, iff) P_\times is safe.

Secure, Precise, and Fast Floating-Point Operations on x86 Processors

Ashay Rane, Calvin Lin

*Department of Computer Science
The University of Texas at Austin
{ashay, lin} @cs.utexas.edu*

Mohit Tiwari

*Dept. of Electrical and Computer Engineering
The University of Texas at Austin
tiwari@austin.utexas.edu*

Abstract

Floating-point computations introduce several side channels. This paper describes the first solution that closes these side channels while preserving the precision of non-secure executions. Our solution exploits microarchitectural features of the x86 architecture along with novel compilation techniques to provide low overhead.

Because of the details of x86 execution, the evaluation of floating-point side channel defenses is quite involved, but we show that our solution is secure, precise, and fast. Our solution closes more side channels than any prior solution. Despite the added security, our solution does not compromise on the precision of the floating-point operations. Finally, for a set of microkernels, our solution is an order of magnitude more efficient than the previous solution.

1 Introduction

To secure our computer systems, considerable effort has been devoted to techniques such as encryption, access control, and information flow analysis. Unfortunately, these mechanisms can often be subverted through the use of side channels, in which an adversary, with the knowledge of the program, monitors the program’s execution to infer secret values. These side channels are significant because they have been used to discover encryption keys in AES [26], RSA [27], and the Diffie-Hellman key exchange protocol [14], thereby rendering these sophisticated schemes useless.

Numerous side channels exist, including instruction and data caches [27, 26], branch predictors [2], memory usage [12, 35], execution time [31, 4], heat [22], power [15], and electromagnetic radiation [9], but one particularly insidious side channel arises from the execution of variable-latency floating-point instructions [3, 10], in which an instruction’s latency varies widely depending on its operands, as shown in Table 1.

Zero	Normal	Subnormal	Infinity	NaN
7	11	153	7	7

Table 1: Latency (in cycles) of the SQRTSS instruction for various operands.

Both x86¹ and ARM² provide variable-latency floating-point instructions. This variable latency stems from the desire to have graceful floating-point arithmetic behavior, which, as we explain in Section 3, requires the use of so-called *subnormal* values [8], which are processed using special algorithms. Since subnormal values are rare, hardware vendors typically support such values in microcode, so as not to slow down the common case. The resulting difference in instruction latency creates a timing side channel, which has been used to infer cross-origin data in browsers and to break differential privacy guarantees of a remote database [3].

However, variable latency floating-point instructions represent only a part of the problem, since higher level floating-point operations, such as sine and cosine, are typically implemented in software. Thus, the implementation of these floating-point operations can leak secret information through other side channels as well. Depending on the secret values, programs can throw exceptions, thereby leaking the presence of abnormal inputs through termination. Programs can also contain conditional branches, which can leak secrets through the instruction pointer, branch predictor, or memory access count. Finally, programs that index into lookup tables can leak secrets through the memory address trace.

To prevent information leaks in both floating-point instructions and floating-point software, a strong solution should ensure at least four key properties, which correspond to the side channels that we discussed above:

¹http://www.agner.org/optimize/instruction_tables.pdf

²<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/ch16s07s01.html>

(1) fixed-time operations that are independent of secret values, (2) disabled exceptions, (3) sequential control flow, and (4) uniform data accesses that are independent of the value of secret variables. Previous solutions [3, 5] are inadequate because they do not ensure all four properties, are slow, are orders of magnitude less precise, or are difficult to implement.

This paper presents a novel solution that closes side channels arising from both hardware and software implementations of floating point operations, providing all four properties mentioned above. Our compiler-based solution has two components.

The first component creates building blocks of elementary floating-point operations for instructions that are natively supported by the hardware (addition, subtraction, multiplication, division, square root, and type conversion). Our solution leverages unused SIMD lanes so that fast operations on normal operands are accompanied by slower dummy computations on subnormal operands, yielding a consistent yet low instruction latency for all types of operands.

The second component is a software library of higher-level floating-point operations like sine and cosine. The key to creating this second component is a new code transformation that produces fixed-latency functions through normalized control flows and data access patterns. Code generated by our compiler closes *digital side-channels*, which have been defined to be those side channels that carry information over discrete bits [28]. Whereas previous work in closing digital side channels employs a runtime system [28], our solution shifts much of the work to compile time, yielding a significantly smaller runtime overhead.

This paper makes the following contributions:

1. We present a novel compiler-based system, called *Escort*, for closing digital side channels that arise from the processing of floating-point instructions.
2. **Secure:** We demonstrate that our solution is secure not just against timing but also against digital side channels. We demonstrate *Escort*'s capabilities by defeating a machine-learning side-channel attack, by defending against a timing attack on the Firefox web browser, by conducting extensive performance measurements on an x86 processor, and by verifying our solution's code using typing rules.
3. **Precise:** We show that *Escort* provides precision that is *identical* to that of the standard C math library. By contrast, the previous solution's precision is off by several million floating-point values.
4. **Fast:** We show that our solution is fast. On a set of micro-benchmarks that exercise elementary

floating-point operations, *Escort* is 16× faster than the previous solution [3].

5. As an ancillary contribution, we introduce a methodology for evaluating the precision and security of floating-point operations, which is fraught with subtleties.

The rest of this paper is organized as follows. Section 2 describes our threat model, related work, and system guarantees. We provide background in Section 3 before presenting our solution in Section 4. We evaluate our solution in Sections 5–7. Finally, we conclude in Section 8.

2 Threat Model and Related Work

This section begins by describing our threat model, which shapes our subsequent discussion of related work and of *Escort*'s security guarantees.

Threat Model. Our goal is to prevent secret floating-point operands from leaking to untrusted principals that either read digital signals from the processor's pins or that are co-resident processes.

We assume that the adversary is either an external entity that monitors *observation-based* side channels (*e.g.* time [14], memory address trace [11], or the /proc pseudo-filesystem [12]) or a co-resident process/VM that monitors *contention-based* side channels (*e.g.* cache [27] or branch predictor state [2]).

For off-chip observation-based channels, we assume that the processor resides in a sealed and tamper-proof chip that prevents the adversary from measuring physical side channels like heat, power, electromagnetic radiation, etc. We assume that the CPU encrypts data transferred to and from DRAM. All components other than the processor are untrusted, and we assume that the adversary can observe and tamper with any digital signal. For on-chip contention-based channels, we assume that the OS is trusted and does not leak the victim process's secret information. We also assume that the adversary cannot observe or change the victim process's register contents. Our trusted computing base includes the compilation toolchain.

Side-Channel Defenses. Decades of prior research have produced numerous defenses against side channels, the vast majority of which close only a limited number of side channels with a single solution. For instance, numerous solutions exist that close only the cache side channel [6, 36, 39, 37, 16] or only the address-trace side channel [33, 20, 32, 29]. Raccoon [28] is the first solution that closes a broad class of side channels—in

particular, the set of digital side channels—with a single solution. Similar to Raccoon, Escort also closes digital side channels with a single solution, but unlike Raccoon, Escort focuses on closing floating-point digital side channels, which can arise from variable latency floating-point instructions and from software implementations of floating-point libraries, in which points-to set sizes are typically small. Given Escort’s narrower focus on floating-point computations, Escort is faster than Raccoon by an order of magnitude.

Timing Side-Channel Defenses. Prior defenses against timing side-channel attacks utilize new algorithms [30], compilers [23], runtime systems [21], or secure processors [18]. However, these solutions only address one source of timing variations—either those stem from the choice of the algorithm [31] or those that stem from the microarchitectural design [10]. By contrast, Escort closes timing variations from both sources.

Floating-Point Side-Channel Defenses. Andryscy et al. [3] present *libfixedtimefixedpoint* (FTFP), the first software solution for closing the floating-point timing channel. FTFP has some weaknesses, as we now discuss, but the main contribution of their paper is the demonstration of the significance of this side channel, as they use variable-latency floating-point operations to break a browser’s same-origin policy and to break differential privacy guarantees of remote databases. FTFP is a *fixed-point* library that consists of 19 hand-written functions that each operates in fixed time, independent of its inputs. FTFP is slow, it is imprecise, and it exposes secrets through other side channels, such as the cache side channel or the address trace side channel. Cleempot et al. [5] introduce compiler transformations that convert variable-timing code into fixed-timing code. Their technique requires extensive manual intervention, applies only to the division operation, and provides weak security guarantees. Both solutions require manual construction of fixed-time code—a cumbersome process that makes it difficult to support a large number of operations. By contrast, Escort implements a fixed-time floating-point library, while preventing information leaks through timing as well as digital side channels. Escort includes a compiler that we have used to automate the transformation of 112 floating-point functions in the Musl standard C library, a POSIX-compliant C library. Escort also provides precision identical to the standard C library.

Escort’s Guarantees. Escort rejects programs that contain unsupported features—I/O operations and recursive function calls. Unlike prior work [18, 28], Escort

does transform loops that leak information through trip counts. Escort is unable to handle programs containing irreducible control flow graphs (CFGs), but standard compiler transformations [24] can transform irreducible CFGs into reducible CFGs. Escort assumes that the input program does not use vector instructions, does not exhibit undefined behavior, does not terminate abnormally through exceptions, and is free of race conditions. Given a program that abides by these limitations, Escort guarantees that the transformed code produces identical results as the original program, does not leak secrets through timing or digital side channels, and that the transformed code does not terminate abnormally.

3 Background

The variable latency of floating-point instructions creates security vulnerabilities. In this section, we explain subnormal numbers, which are the cause of the variable latency, and we explain the difficulty of fixing the resulting vulnerability. We also explain how the Unit of Least Precision (ULP) can be used to quantify the precision of our and competing solutions.

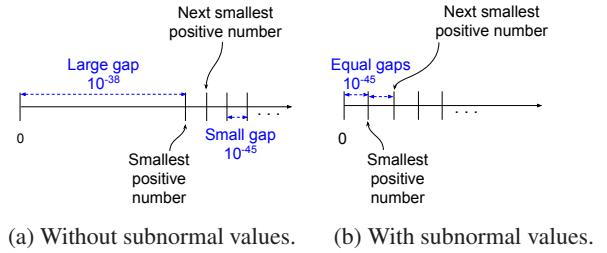


Figure 1: Impact of allowing subnormal numbers. Without subnormal values, there exists a much larger gap between zero and the smallest positive number than between the first two smallest positive numbers. With subnormal numbers, the values are more equally spaced. (The figure is not drawn to scale.)

3.1 Subnormal Numbers

Subnormal numbers have tiny exponents, which result in floating-point values that are extremely close to zero: $10^{-45} < |x| < 10^{-38}$ for single-precision numbers and $10^{-324} < |x| < 10^{-308}$ for double-precision numbers. Subnormal values extend the range of floating-point numbers that can be represented, but more importantly, they enable *gradual underflow*—the property that as floating-point numbers approach zero along the number scale, the difference between successive floating-point numbers does not increase³. Figures 1a and 1b show the

³https://www.cs.berkeley.edu/~wkahan/ARITH_17U.pdf

differences between zero and the two smallest positive floating-point numbers. With subnormal numbers, the gap between any two consecutive floating-point values is never larger than the values themselves, thus exhibiting Gradual Underflow. Subnormal numbers are indispensable because gradual underflow is required for reliable equation solving and convergence acceleration [8, 13].

To avoid the added hardware complexity of supporting subnormal numbers, which occur infrequently, vendors typically process subnormal values in microcode, which is orders of magnitude slower than hardwired logic.

The resulting difference in latencies creates a security vulnerability. An adversary that can measure the latency of a floating-point instruction can make reasonable estimates about the operand type, potentially inferring secret values using the timing channel. While subnormal values occur infrequently in typical program execution, an adversary can deliberately induce subnormal values in the application’s inputs to enable subnormal operand timing attacks.

3.2 Floating-Point Error Measurement

Unlike real (infinite precision) numbers, floating-point numbers use a limited number of bits to store values, thus making them prone to rounding errors. Rounding errors in floating-point numbers are typically measured in terms of the Unit of Least Precision (ULP) [25]. The ULP distance between two floating-point numbers is the number of distinct representable floating-point numbers between them, which is simply the result of subtracting their integer representations. If the result of the subtraction is zero, the floating-point numbers must be exactly the same.

4 Our Solution: Escort

Escort offers secure counterparts of ordinary non-secure floating-point operations, including both elementary operations and higher-level math operations. The elementary operations include the six basic floating-point operations that are natively supported by the ISA—type conversion, addition, subtraction, multiplication, division, and square root—and a conditional data copy operation. The 112 higher-level math operations are those that are implemented using a combination of native instructions. Examples of higher-level functions include sine, cosine, tangent, power, logarithm, exponentiation, absolute value, floor, and ceiling.

The next subsections describe Escort’s design in three parts. First, we describe the design of Escort’s secure elementary operations. These operations collectively form the foundation of Escort’s security guarantees. Second, we describe Escort’s compiler, which accepts non-secure

code for higher-level operations and converts it into secure code. This compiler combines a code transformation technique with Escort’s secure elementary operations. Third, we present an example that shows the synergy among Escort’s components.

4.1 Elementary Operations

The key insight behind Escort’s secure elementary operations is that the latencies of SIMD instructions are determined by the slowest operation among the SIMD lanes (see Figure 2), so the Escort compiler ensures that each elementary instruction runs along side a dummy instruction whose operand will produce the longest possible latency. Our analysis of 94 x86 SSE and SSE2 instructions (which includes single- and double-precision arithmetic, comparison, logical, and conversion instructions) reveals: (1) that only the multiplication, division, square root, and single-precision to double-precision conversion (upcast) instructions exhibit latencies that depend on their operands and (2) that subnormal operands induce the longest latency.

In particular, Escort’s fixed-time floating-point operations utilize SIMD lanes in x86 SSE and SSE2 instructions. Our solution (1) loads genuine and dummy (subnormal) inputs in spare SIMD lanes of the same input register, (2) invokes the desired SIMD instruction, and (3) retains only the result of the operation on the genuine inputs. Our tests confirm that the resulting SIMD instruction exhibits the worst-case latency, with negligible variation in running time (standard deviation is at most 1.5% of the mean). Figure 3 shows Escort’s implementation of one such operation.

Escort includes Raccoon’s conditional data copy operation (see Figure 4) which does not leak information through digital side channels. This operation copies the contents of one register to another register if the given condition is true. However, regardless of the condition, this operation consumes a fixed amount of time, executes the same set of instructions, and does not access application memory.

4.2 Compiling Higher-Level Operations

Escort’s compiler converts existing non-secure code into secure code that prevents information leakage through digital side channels. First, our compiler replaces all elementary floating-point operations with their secure counterparts. Next, our compiler produces straight-line code that preserves control dependences among basic blocks while preventing instruction side effects from leaking secrets. Our compiler then transforms array access statements so that they do not leak information through memory address traces. Finally, our compiler transforms

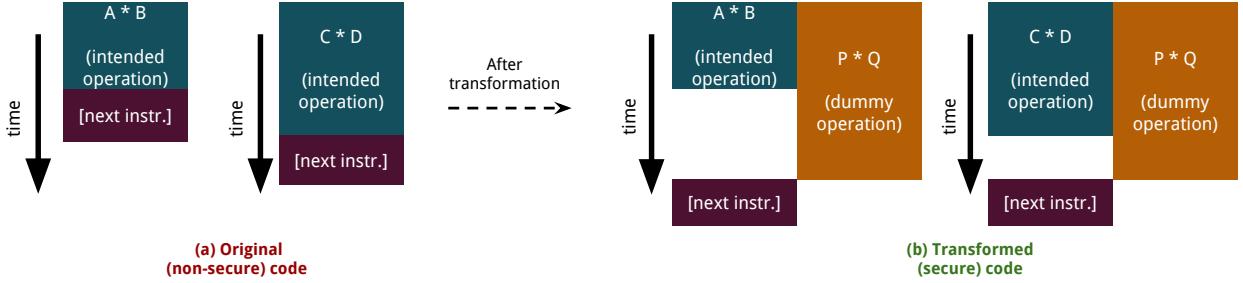


Figure 2: The key idea behind Escort’s secure elementary operations. The operation is forced to exhibit a fixed latency by executing a fixed-latency long-running operation in a spare SIMD lane.

```
double escort_mul_dp(double x, double y) {
    const double k_normal_dp = 1.4;
    const double k_subnormal_dp = 2.225e-322;

    double result;
    __asm__ volatile(
        "movdqa    %1, %%xmm14;"          // Load x
        "movdqa    %2, %%xmm15;"          // Load y
        "pslldq   $8, %1;"              // Shift x left by 8
        "pslldq   $8, %2;"              // Shift y left by 8
        "por      %3, %1;"              // Compute sign of result
        "por      %4, %2;"              // Compute sign of result
        "movdqa   %2, %0;"              // Load k_subnormal_dp
        "mulpd   %1, %0;"              // Compute result
        "psrldq   $8, %0;"              // Shift result right by 8
        "movdqa   %%xmm14, %1;"         // Store result
        "movdqa   %%xmm15, %2;"         // Store y
        : "=x" (result), "+x" (x), "+x" (y)
        : "x" (k_subnormal_dp), "x" (k_normal_dp)
        : "xmm15", "xmm14");
    return result;
}
```

Figure 3: Escort’s implementation of double-precision multiplication, using the AT&T syntax.

loops whose trip count reveals secrets over digital side channels. We now describe each step in turn.

4.2.1 Step 1: Using Secure Elementary Operations

The Escort compiler replaces x86 floating-point type-conversion, multiplication, division, and square root assembly instructions with their Escort counterparts. However, Escort’s secure elementary operations can be up to two orders of magnitude slower than their non-secure counterparts. Hence, our compiler minimizes their usage by using taint tracking and by employing the quantifier-free bit-vector logic in the Z3 SMT solver [7], which is equipped with floating-point number theory. If the solver can prove that the operands can never be subnormal values, then Escort refrains from replacing that instruction.

In effect, the Escort compiler constructs path-sensitive Z3 expressions for each arithmetic statement in the

```
01: copy(uint8_t pred, uint32_t t_val, uint32_t f_val) {
02:     uint32_t result;
03:     __asm__ volatile (
04:         "mov    %2, %0;"           // Move t_val to result
05:         "test   %1, %1;"          // Test pred
06:         "cmovz %3, %0;"          // If pred is true, move f_val to result
07:         "test   %2, %2;"          // Test pred again
08:         : "=r" (result)
09:         : "r" (pred), "r" (t_val), "r" (f_val)
10:         : "cc"
11:     );
12:     return result;
13: }
```

Figure 4: Code for conditional data copy operation that does not leak information over digital side channels. This function returns `t_val` if `pred` is `true`; otherwise it returns `f_val`. The assembly code uses AT&T syntax.

LLVM IR. For every Φ -node that produces an operand for an arithmetic expression, Escort creates one copy of the expression for each input to the Φ -node. If the solver reports that no operand can have a subnormal value, then Escort skips instrumentation of that floating-point operation.

We set a timeout of 40 seconds for each invocation of the SMT solver. If the solver can prove that the instruction never uses subnormal operands, then Escort skips replacing that floating-point instruction with its secure counterpart. Figure 5 shows the percentage of floating-point instructions in commonly used math functions that are left untransformed by Escort.

This optimization is conservative because it assumes that all floating-point instructions in the program have subnormal operands unless proven otherwise. The correctness of the optimization is independent of the code’s use of pointers, library calls, system calls, or dynamic values. The static analysis used in this optimization is flow-sensitive, path-sensitive, and intra-procedural.

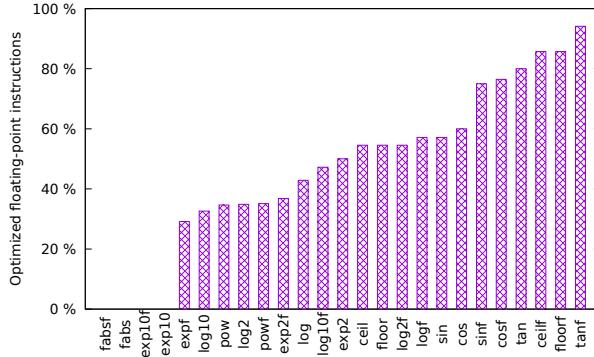


Figure 5: Percentage of instructions that are left uninstrumented (without sacrificing security) after consulting the SMT solver.

4.2.2 Step 2: Predicating Basic Blocks

Basic block predicates represent the conditions that dictate whether an instruction should execute. These predicates are derived by analyzing conditional branch instructions. For each conditional branch instruction that evaluates a predicate p , the Escort compiler associates the predicate p with all basic blocks that execute if the predicate is true, and it associates the predicate $\neg p$ with all basic blocks that execute if the predicate is false. For unconditional branches, the compiler copies the predicate of the previous block into the next block. Finally, if the Escort compiler comes across a block that already has a predicate, then the compiler sets the block’s new predicate to the logical OR of the input predicates. At each step, the Escort compiler uses Z3 as a SAT solver to simplify predicates by eliminating unnecessary variables in predicate formulas. Figure 6 shows the algorithm for basic block predication.

4.2.3 Step 3: Linearizing Basic Blocks

The Escort compiler converts the given code into straight-line code so that every invocation of the code executes the same instructions. To preserve control dependences, the basic blocks are topologically sorted, and then the code is assembled into a single basic block with branch instructions removed.

4.2.4 Step 4: Controlling Side Effects

We now explain how Escort prevents side effects from leaking secrets. Here, *side effects* are modifications to the program state or any observable interaction, including memory accesses, exceptions, function calls, or I/O. Escort controls all side effects except for I/O statements.

```

1: for each basic block  $bb$  in function do
2:   if  $entry\_block(bb)$  then
3:      $pred[bb] \leftarrow \text{true}$ 
4:   else
5:      $pred[bb] \leftarrow \text{false}$ 
6:   end if
7: end for
8:
9: for each basic block  $bb$  in function do
10:   $br \leftarrow branch(bb)$ 
11:  if  $unconditional\_branch(br)$  then
12:     $\{s\} \leftarrow successors(bb)$ 
13:     $pred[s] \leftarrow pred[s] \vee pred[bb]$ 
14:     $pred[s] \leftarrow simplify(pred[s])$ 
15:  else                                 $\triangleright$  Conditional Branch.
16:     $\{s_1, s_2\} \leftarrow successors(bb)$ 
17:    if  $loop\_condition\_branch(br)$  then
18:       $\triangleright$  Skip branches that represent loops.
19:       $pred[s_1] \leftarrow pred[s_1] \vee pred[bb]$ 
20:       $pred[s_2] \leftarrow pred[s_2] \vee pred[bb]$ 
21:    else
22:       $p \leftarrow condition(br)$ 
23:       $pred[s_1] \leftarrow pred[s_1] \vee (pred[bb] \wedge p)$ 
24:       $pred[s_2] \leftarrow pred[s_2] \vee (pred[bb] \wedge \neg p)$ 
25:    end if
26:     $pred[s_1] \leftarrow simplify(pred[s_1])$ 
27:     $pred[s_2] \leftarrow simplify(pred[s_2])$ 
28:  end if
29: end for

```

Figure 6: Algorithm for predication basic blocks.

Memory Access Side Effects. To ensure proper memory access side effects, the Escort compiler replaces store instructions with conditional data-copy operations that are guarded by the basic block’s predicate, so memory is only updated by instructions whose predicate is true.

Unfortunately, this naïve approach can leak secret information when the program uses pointers. Figure 7 illustrates the problem: If store instructions are not allowed to update a pointer variable when the basic block predicate is false, then the address trace from subsequent load instructions on the pointer variable will expose the fact that the pointer variable was not updated.

The Escort compiler prevents such information leaks by statically replacing pointer dereferences with loads or stores to each element of the points-to set⁴. Thus Escort replaces the statement in line 8 (Figure 7) with a store operation on b . When the points-to set is larger than a

⁴Escort uses a flow-sensitive, context-insensitive pointer analysis: <https://github.com/grievejia/tpa>. Replacing a pointer dereference with a store operation on all elements of the points-to set is feasible for Escort because points-to set sizes in the Musl C library are very small.

```

1:  $p \leftarrow \&a$ 
2:  $secret \leftarrow input()$      $\triangleright$  Assume  $input()$  returns true.
3: if  $secret = \text{true}$  then
4:   ...
5: else
6:   ...
7:      $p \leftarrow \&b$   $\triangleright$  Instruction does not update pointer  $p$ ,
       since basic block's execution-time predicate is false.
8:      $*p \leftarrow 10$             $\triangleright$  Accesses  $a$  instead of  $b$ !
9: end if

```

Figure 7: The use of pointers can leak information. If store instructions are not allowed to access memory when the basic block’s predicate is false, then pointer p will dereference the address for a instead of b , thus revealing that $secret$ is true.

singleton set, Escort uses the conditional data copy operation on all potential *pointees* *i.e.* the elements of the points-to set. The predicate of the conditional copy operation checks whether the pointer points to the candidate pointee. If the predicate is false, the pointee’s existing value is overwritten, whereas if the predicate is true, the new value is written to the pointee.

Function Call Side Effects. Adversaries can observe the invocation of functions (or lack thereof) using side channels like the Instruction Pointer. Thus, a solution incapable of handling function calls will leak information to the adversary. While inlining functions is a potential solution, inlining is impractical for large applications.

Escort handles side effects from function calls by propagating the predicate from the calling function to the callee. Thus, each user-defined function is given an additional argument that represents the predicate of the call site’s basic block. The callee ensures correct handling of side effects by ANDing its own predicates with the caller’s predicate.

Side Effects from Exceptions. Program termination caused by exceptions will leak the presence or absence of abnormal operands. To prevent such information leakage, Escort requires that exceptions not occur during program execution⁵.

Escort manages floating-point and integer exceptions differently. Escort requires that the programmer disable floating-point exceptions (*e.g.* using `feclearexcept()`). For integer exceptions, Escort borrows ideas from Raccoon by replacing abnormal operands with benign operands (*e.g.* Escort prevents integer division-by-zero by replacing a zero divisor with a non-zero divisor).

⁵Escort assumes that the input program does not throw exceptions, so masking exceptions does not change the semantics of the program.

4.2.5 Step 5: Transforming Array Accesses

Array index values reveal secrets as well. For instance, if the adversary observes that accesses to $array[0]$ and $array[secret_index]$ result in accesses to locations 10 and 50, then the adversary knows that $secret_index = 40$. To eliminate such information leaks, the Escort compiler transforms each array access into a linear sweep over the entire array, which hides from the adversary the address of the program’s actual array index.

Of course, the transformed code is expensive, but this approach is feasible because (1) math library functions typically use only a few small lookup tables, thus requiring relatively few memory accesses and (2) the processor’s caches and prefetchers dramatically reduce the cost of sweeping over the arrays.

4.2.6 Step 6: Transforming Loops

Some loops introduce timing channels because their trip counts depend on secret values. The Escort compiler transforms such loops using predictive mitigation [38]. The loop body executes as many times as the smallest power of 2 that is greater than or equal to the loop trip count. For instance, if the actual loop trip count is 10, then the loop body is executed 16 times. The basic block predicate ensures that dummy iterations do not cause side effects. With this transformed code, an adversary that observes a loop trip count of l can infer that the actual loop trip count l' is between l and $0.5 \times l$. However, the exact value of l' is not revealed to the adversary.

Unfortunately, this naive approach can still leak information. For instance, if two distinct inputs cause the loop to iterate 10 and 1000 times respectively, the transformed codes will iterate 16 and 1024 times respectively—a large difference that may create timing variations. To mitigate this problem, Escort allows the programmer to manually specify the minimum and maximum loop trip counts using programmer annotations. These annotations override the default settings used by the Escort compiler.

4.3 Example Transformation: `exp10f`

We now explain how Escort transforms an example non-secure function (Figure 8a) into a secure function (Figure 8c). To simplify subsequent analyses and transformations, the Escort compiler applies LLVM’s `mergereturn` transformation pass, which unifies all exit nodes in the input function (see Figure 8b).

First, the Escort compiler replaces elementary floating-point operations in lines 8 and 10 with their secure counterpart function shown in lines 21 and 22 of the transformed code. Second, using the algorithm shown in Figure 6, the Escort compiler associates predicates with

```

float e10(float x) {
    float n, y = mf(x, &n);
    if (int(n) >> 23 & 0xff < 0x82) {
        float p = p10[(int) n + 7];
        if (y == 0.0f) {
            return p;
        }
        return exp2f(3.322f * y) * p;
    }
    return exp2(3.322 * x);
}

```

(a) Original code for `exp10f()`.

```

01: float e10(float x) {
02:     float n, y = mf(x, &n);
03:     if (int(n) >> 23 & 0xff < 0x82) {
04:         float p = p10[(int) n + 7];
05:         if (y == 0.0f)
06:             result = p;
07:         else
08:             result =
09:                 exp2f(3.322f * y) * p;
10:     } else
11:         result = exp2(3.322 * x);
12:     return result;
}

```

(b) Result after applying LLVM's `mergereturn` pass. This code becomes the input for the Escort compiler.

```

12: float e10(float x) {
13:     return e10_cloned(x, true);
14: }
15:
16: float e10_cloned(float x, uint pred) {
17:     float n, y = mf_cloned(x, &n, pred);
18:     float p = write(int(n) >> 23 & 0xff
19:                     < 0x82, stream_load(p10, (int) n + 7));
20:     bool p2 = y == 0.0f;
21:     write(pred & p1 & p2, p, &result);
22:     write(pred & p1 & !p2,
23:           escort_mul(
24:             escort_mul(
25:               exp2f_cloned(3.322f,
26:                             pred & p1 & !p2),
27:               y),
28:               p),
29:             &result);
30:
31:     write(!p1,
32:           escort_mul(
33:             exp2_cloned(3.322, pred & !p1),
34:             escort_upcast(x))),
35:             result);
36:
37:     return result;
38: }

```

(c) Result of the Escort compiler's transformation.

Figure 8: Escort's transformation of `exp10f()`.

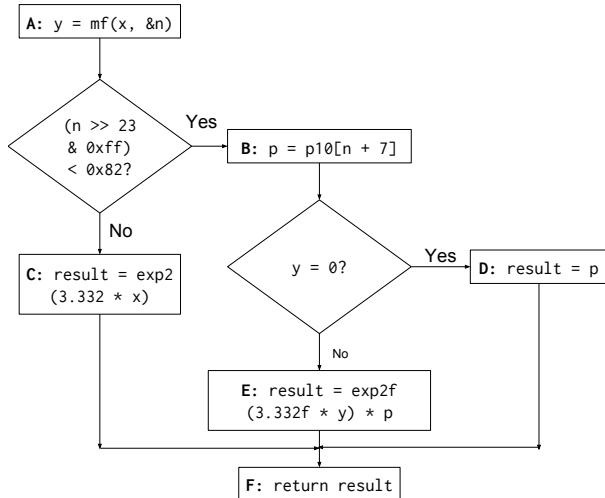


Figure 9: Control flow graph with labeled statements for the code in Figure 8b. **A, B, D, E, C, F** is one possible sequence of basic blocks when linearized by the Escort compiler.

Line #	Predicate
2, 3, 11	TRUE
4, 5	(n >> 23 & 0xff) < 0x82
6	(n >> 23 & 0xff) < 0x82 \wedge y = 0
8	(n >> 23 & 0xff) < 0x82 \wedge y \neq 0
10	$\neg((n >> 23 & 0xff) < 0x82)$

Table 2: Predicates per line for function in Figure 8b.

each basic block, which we list in Table 2. Third, the Escort compiler linearizes basic blocks by applying a topological sort on the control flow graph (see Figure 9) and fuses the basic blocks together. Finally, the Escort compiler replaces the array access statement in line 4 with a function that sweeps over the entire array. The resulting code, shown in Figure 8c, eliminates control flows and data flows that depend on secret values. In addition to closing digital side channels, the code also uses secure floating-point operations.

5 Security Evaluation

This section demonstrates that Escort’s floating-point operations run in fixed time and do not leak information through digital side channels. Since precise timing measurement on x86 processors is tricky due to complex processor and OS design, we take special measures to ensure that our measurements are accurate. In addition to Escort’s timing and digital side channel defense, we also demonstrate Escort’s defense against a floating-point timing channel attack on the Firefox web browser.

5.1 Experimental Setup

We run all experiments on a 4-core Intel Core i7-2600 (Sandy Bridge) processor. The processor is clocked at 3.4 GHz. Each core on this processor has a 32 KB private L1 instruction cache, a 32 KB private L1 data cache, and a 256 KB private L2 cache. A single 8 MB L3 cache is shared among all four cores. The host operating system is Ubuntu 14.04 running kernel version 3.13. We implement compiler transformations using the LLVM compiler framework [17] version 3.8.

We measure instruction latencies using the RDTSC instruction that returns the number of elapsed cycles since resetting the processor. Since the latency of executing the RDTSC instruction is usually higher than the latency of executing operations, our setup measures the latency of executing 1024 consecutive operations and divides the measured latency by 1024. Our setup uses the CPUID instruction and volatile variables for preventing the processor and the compiler from reordering critical instructions. Finally, our setup measures overhead by executing an empty loop body—a loop body that contains no instructions other than those in the test harness. By placing an empty volatile `__asm__` block in the empty loop body, our setup prevents the compiler from deleting the empty loop body.

5.1.1 Outlier Elimination

Many factors outside of the experiment’s control, like interrupts, scheduling policies, etc., may result in outliers in performance measurements. We now explain our procedure for eliminating outliers, before demonstrating that the elimination of these outliers does not bias the conclusions.

We use Tukey’s method [34] for identifying outliers, but we adapt it to conservatively classify fewer values as outliers (thus including more values as valid data points). The original Tukey’s method first finds the minimum (M_n), median (M_d), and maximum (M_x) of a set of values. The first quartile, Q_1 , is the median of values between M_n and M_d . The third quartile, Q_3 , is the median of values

between M_x and M_d . The difference between the first and the third quartiles ($Q_3 - Q_1$) is called the Inter-Quartile Range, R_{IQ} . Tukey’s method states that any value v , such that $v > Q_3 + 3 \times R_{IQ}$ or $v < Q_1 - 3 \times R_{IQ}$ is a probable outlier. In our evaluation, we weaken our outlier elimination process (*i.e.* we count fewer values as outliers), by (1) setting the R_{IQ} to be at least equal to 1.0, and (2) classifying v as an outlier when $v > Q_3 + 20 \times R_{IQ}$ or $v < Q_1 - 20 \times R_{IQ}$. Results presented in the following sections use the relaxed Tukey method described above.

	Mean	Median	Std. Dev.
Different Operands	847,323 (0.81%)	1,066,270 (1.02%)	381,467
Same Operands	929,703 (0.89%)	1,139,961 (1.09%)	364,192

Table 3: Number of discarded outliers from 100 million double-precision square-root operations. The results indicate that our outlier elimination process is statistically independent of the input operand values.

To demonstrate that our outlier elimination process does not bias conclusions, we compare the distribution of outliers between (a) 100 million operations using randomly-generated operands, and (b) 100 million operations using one fixed operand. The two experiments do not differ in any way other than the difference in their input operands. Table 3 shows the mean, median, and standard deviation of outliers for the double-precision square-root operation. Results for other floating-point operations are similar and are elided for space reasons. Since the difference in mean values as well as the difference in median values is within a quarter of the standard deviation from the mean, we conclude that the discarded outlier count is statistically independent of the input operand values.

5.2 Timing Assurance of Elementary Operations

Since exhaustively testing all possible inputs for each operation is infeasible, we instead take the following three-step approach for demonstrating the timing channel defense for Escort’s elementary operations: (1) We characterize the performance of Escort’s elementary operations using a specific, fixed floating-point value (*e.g.* 1.0), (2) using one value from each of the six different types of values (zero, normal, subnormal, $+\infty$, $-\infty$, and not-a-number), we show that our solution exhibits negligible variance in running time, and (3) to demonstrate that each of the six values in the previous experiment is representative of the class to which it belongs, we generate 10 million normal, subnormal, and not-a-number (NaN)

values, and show that the variance in running time among each set of 10 million values is negligible. Our key findings are that Escort’s operations run in fixed time, are fast, and that their performance is closely tied to the performance of the hardware’s subnormal operations.

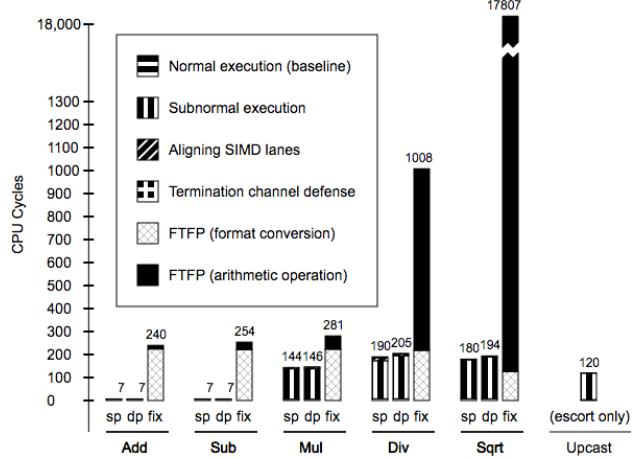


Figure 10: Comparison of running times of elementary operations. **sp** identifies Escort’s single-precision operations, **dp** identifies Escort’s double-precision operations, and **fix** identifies FTFP’s fixed-point operations. Numbers at the top of the bars show the total cycle count. We see that Escort’s execution times are dominated by the cost of subnormal operations, and we see that FTFP’s overheads are significantly greater than Escort’s.

Figure 10 compares the running times of elementary operations of Escort and of previous solutions (FTFP). First, we observe that the running times of Escort’s single- and double-precision operations are an order-of-magnitude lower than those of FTFP’s fixed-precision operations. Second, Escort’s running time is almost entirely dominated by the processor’s operation on subnormal numbers. Third, conversion between fixed-point and floating-point takes a non-trivial amount of time, further increasing the overhead of FTFP’s operations. Overall, Escort elementary operations are about 16× faster than FTFP’s.

Table 4 shows the variation in running time of elementary operations across six different types of inputs (zero, normal value, subnormal value, $+\infty$, $-\infty$, and not-a-number value) and compares it with the variation of SSE (native) operations. While SSE operations exhibit high variation (the maximum observed standard deviation is 176% of the mean), Escort’s operations show negligible variation across different input types.

Finally, we measure Escort’s running time for 10 million random normal, subnormal, and not-a-number values. We observe that the standard deviation of these measurements, shown in Table 5, is extremely low (at most

Function	Escort	Native (SSE)
add-sp	0	0
add-dp	0	0
sub-sp	0	0
sub-dp	0	0
mul-sp	0	49.2 (175%)
mul-dp	0	49.2 (175%)
div-sp	0.66 (0.4%)	65.67 (163%)
div-dp	1.66 (0.8%)	69.08 (164%)
sqrt-sp	1.49 (0.8%)	62.7 (170%)
sqrt-dp	2.98 (1.5%)	66.87 (169%)
upcast	0	40.99 (178%)

Table 4: Comparison of standard deviation of running times of elementary operations across six types of values (zero, normal, subnormal, $+\infty$, $-\infty$, and not-a-number). Numbers in parenthesis show the standard deviation as a percentage of the mean. The **-sp** suffix identifies single-precision operations while the **-dp** suffix identifies double-precision operations. Compared to SSE operations, Escort exhibits negligible variation in running times.

3.1% of the mean). We thus conclude that our chosen values for each of the six classes faithfully represent their class.

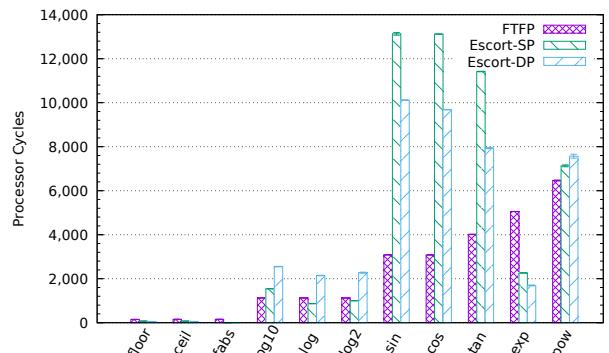


Figure 11: Comparison of running times of commonly used higher-level functions. Error bars (visible for only a few functions) show the maximum variation in running time for different kinds of input values.

5.3 Timing Assurance of Higher-Level Operations

Using different types of floating-point values (zero, normal, subnormal, $+\infty$, $-\infty$, and not-a-number), Figure 11 compares the performance of most of the commonly used

Fn.	NaN	Normal	Subnormal
add-sp	0.21 (3.1%)	0.21 (2.9%)	0.19 (2.7%)
add-dp	0.21 (3.0%)	0.20 (2.9%)	0.21 (3.0%)
sub-sp	0.18 (2.6%)	0.19 (2.7%)	0.20 (2.9%)
sub-dp	0.19 (2.7%)	0.19 (2.7%)	0.19 (2.7%)
mul-sp	0.98 (0.7%)	0.94 (0.7%)	1.05 (0.7%)
mul-dp	0.90 (0.6%)	1.04 (0.7%)	1.02 (0.7%)
div-sp	1.22 (0.6%)	1.27 (0.7%)	1.23 (0.6%)
div-dp	1.39 (0.7%)	1.37 (0.6%)	1.17 (0.6%)
sqrt-sp	1.15 (0.6%)	1.13 (0.6%)	1.14 (0.6%)
sqrt-dp	1.29 (0.7%)	1.41 (0.7%)	1.33 (0.7%)
upcast	1.03 (0.9%)	0.89 (0.8%)	0.95 (0.8%)

Table 5: Standard deviation of 10 million measurements for each type of value (normal, subnormal, and not-a-number). All standard deviation values are within 3.1% of the mean. Furthermore, the mean of these 10,000,000 measurements is always within 2.7% of the representative measurement.

single- and double-precision higher-level operations⁶. Overall Escort’s higher-level operations are about $2\times$ slower than their corresponding FTFP operation, which is the price for closing side channels that FTFP does not close.

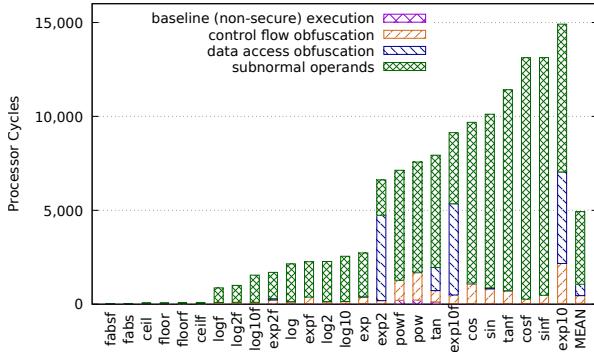


Figure 12: Performance breakdown of Escort’s commonly used higher-level functions. The baseline (non-secure) execution and exception handling together cost less than 250 cycles for each function, making them too small to be clearly visible in the above plot.

Figure 12 shows the breakdown of the performance of commonly used higher-level functions. We observe that the performance of most higher-level functions is dominated by the latency of operations on subnormal operands, which is closely tied to the performance of the underlying hardware. A handful of routines ($\text{exp10}()$,

⁶We exclude the $\text{exp2}()$ (6,617 cycles), $\text{exp10}()$ (14,910 cycles), $\text{exp2f}()$ (1,693 cycles), and $\text{exp10f}()$ (9,134 cycles) from Figure 11 because FTFP does not implement these operations.

$\text{exp10f}()$, $\text{exp2f}()$, and $\text{exp2f}()$ use lookup tables that are susceptible to address-trace-based side-channel information leaks, so the code transformed by Escort sweeps over these lookup tables for each access to the table. Finally, we see that the cost of control flow obfuscation (*i.e.* the cost of executing all instructions in the program) contributes the least to the total overhead.

5.4 Side-Channel Defense in Firefox

We now evaluate Escort’s defense against the timing channel attack by Andryesco et al. [3] on the Firefox web browser. The attack reconstructs a two-color image inside a victim web page using only the timing side channel in floating-point operations. The attack convolves the given secret image with a matrix of subnormal values. The convolution step for each pixel is timed using high resolution Javascript timers. By comparing the measured time to a threshold, each pixel is classified as either black or white, effectively reconstructing the secret image.

We integrate Escort into Firefox’s convolution code⁷ and re-run the timing attack. The results (see Figure 13c) show that Escort successfully disables the timing attack.

5.5 Control- and Data-Flow Assurance

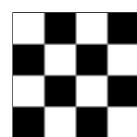
We now show that Escort’s operations do not leak information through control flow or data flow. We first use inference rules over the LLVM IR to demonstrate non-interference between secret inputs and digital side channels. We run a machine-learning attack on Escort and demonstrate that Escort successfully disables the attack.

5.5.1 Non-Interference Using Inference Rules

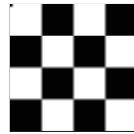
Since Escort’s elementary operations are small and simple—they are implemented using fewer than 15 lines of assembly code, they do not access memory, and they do not contain branch instructions—they are easily verified for non-interference between secret inputs and digital side channels. Using an LLVM pass that applies the inference rules from Table 6, tracking labels that can be either **L** (for low-context *i.e.* public information) or **H** (for high-context *i.e.* private information), we verify that Escort’s higher-level operations close digital side channels. This compiler pass initializes all function arguments with the label **H**, since arguments represent secret inputs.

Inference rules for various instructions dictate updates to the labels. The environment Γ tracks the label of each pointer and each address. The Escort compiler tags load

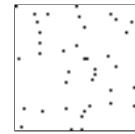
⁷Specifically, we replace three single-precision multiplication operations with invocations to the equivalent Escort function. All source code changes are limited to the code in the `ConvolvePixel()` function in `SVGFEConvolveMatrixElement.cpp`.



(a) Original image.



(b) Reconstructed image using timing attack.



(c) Reconstructed images in 3 independent, consecutive experiments after patching Firefox with Escort.

Figure 13: Results of attack and defense on a vulnerable Firefox browser using timing-channel information leaks arising from the use of subnormal floating-point numbers.

and store instructions as secret if the pointer is tainted, or public otherwise. Unlike a public load or store instruction, a secret load or store instruction is allowed to use a tainted pointer since Escort generates corresponding loads and stores to *all* statically-determined candidate values in the points-to set. The sanitization rule resets the value’s label to L and is required to suppress false alarms from Escort’s loop condition transformation. Escort’s transformed code includes instructions with special LLVM metadata that trigger the sanitization rule.

During verification, the compiler pass iterates over each instruction and checks whether a rule is applicable using the rule’s antecedents (the statement above the horizontal line); if so, it updates its local state as per the rule’s consequent (the statement below the horizontal line). If no applicable rule is found, then the compiler pass throws an error. The compiler pass processes the code for Escort’s 112 higher-level operations without throwing errors.

5.5.2 Defense Against Machine-Learning Attack

We use the TensorFlow [1] library to design a machine-learning classifier, which we use to launch a side-channel attack on the execution of the `expf()` function, where the input to the `expf()` function is assumed to be secret. Using three distinct inputs, we run this attack on the implementations in the (non-secure) Musl C library and in the (secure) Escort library. We first use the Pin dynamic binary instrumentation tool [19] to gather the full instruction address traces of both `expf()` implementations⁸. We train the TensorFlow machine-learning classifier by feeding the instruction address traces to the classifier, associating each trace with the secret input to `expf()`. We use cross entropy as the cost function for TensorFlow’s training phase. In the subsequent testing phase, we randomly select one of the collected address traces and ask the classifier to predict the secret input value.

We find that for the Musl implementation, the classifier is accurately able to predict the correct secret value from the address trace. On the other hand, for the Escort

implementation, the classifier’s accuracy drops to 33%, which is no better than randomly guessing one of the three secret input values.

6 Precision Evaluation

We examine the precision of Escort and FTFP by comparing Escort’s and FTFP’s results with those produced by a standard C library.

6.1 Comparison Using Unit of Least Precision

Methodology. We adopt an empirical approach to estimate precision in terms of Unit of Least Precision (ULP), since formal derivation of maximum ULP difference requires an intricate understanding of theorem provers and floating-point algorithms. We run various floating-point operations on 10,000 randomly generated pairs (using `drand48()`) of floating-point numbers between zero and one. For elementary operations, we compare the outputs of Escort and FTFP with the outputs of native x86 instructions. For all other operations, we compare the outputs of Escort and FTFP with the outputs produced by corresponding function from the Musl C library.

Results. We observe that Escort’s results are identical to the results produced by the reference implementations, *i.e.* the native (x86) instructions and the Musl C library. More precisely, the ULP difference between Escort’s results and reference implementation’s results is zero. On the other hand, FTFP, which computes arithmetic in fixed-point precision, produces output that differs substantially from the output of Musl’s double-precision functions (see Table 7). The IEEE 754 standard requires that addition, subtraction, multiplication, division, and square root operations are computed with ULP difference of at most 0.5. Well-known libraries compute results for most higher-level operations within 1 ULP.

⁸Using the `md5sum` program, we observe that Escort’s address traces for all three inputs are identical.

T-PUBLIC-LOAD	$P = ptset(\text{ptr})$
	$\Gamma(\text{ptr}) = \mathbf{L}$
	$m = \max_{\text{addr} \in P} \Gamma(\text{addr})$
	$\Gamma' = \Gamma[\text{val} \mapsto m]$
	$\frac{}{\Gamma \vdash \text{val} := \text{public-load } \text{ptr} : \Gamma'}$
T-PUBLIC-STORE	$\forall \text{addr} \in ptset(p)$
	$\Gamma(\text{ptr}) = \mathbf{L}$
	$m = \max(\Gamma(\text{val}), \Gamma(\text{addr}))$
	$\Gamma' = \Gamma[\text{addr} \mapsto m]$
	$\frac{}{\Gamma \vdash \text{public-store } \text{ptr}, \text{val} : \Gamma'}$
T-SECRET-LOAD	$\Gamma' = \Gamma[\text{val} \mapsto H]$
	$\frac{}{\Gamma \vdash \text{val} := \text{secret-load } \text{ptr} : \Gamma'}$
T-SECRET-STORE	$\forall \text{addr} \in ptset(p)$
	$\Gamma' = \Gamma[\text{addr} \mapsto H]$
	$\frac{}{\Gamma \vdash \text{secret-store } \text{ptr}, \text{val} : \Gamma'}$
T-BRANCH	$\Gamma(\text{cond}) = \mathbf{L}$
	$\frac{}{\Gamma \vdash \text{br cond,block1,block2} : \Gamma'}$
T-OTHER	$\Gamma' = \Gamma[x \mapsto \Gamma(y)]$
	$\frac{}{\Gamma \vdash x := y : \Gamma'}$
T-COMPOSITION	$\Gamma \vdash S_1 : \Gamma', \Gamma' \vdash S_2 : \Gamma''$
	$\frac{}{\Gamma \vdash S_1; S_2 : \Gamma''}$
T-SANITIZER	$\Gamma' = \Gamma[x \mapsto L]$
	$\frac{}{\Gamma \vdash S(x) : \Gamma'}$

Table 6: Inference rules for verifying the security of Escort’s higher-level operations.

6.2 Comparison of Program Output

Methodology. Since differences in program outputs provide an intuitive understanding of the error introduced by approximate arithmetic operations, we compare the output of the test suite of Minpack⁹, a library for solving non-linear equations and non-linear least squares problems. We generate three variants of Minpack: MINPACK-C uses the standard GNU C library, MINPACK-ESCORT uses the Escort library, and MINPACK-FTFP uses the FTFP library. We run the 29 programs in Minpack’s test suite and compare the outputs produced by the three program variants.

Results. We observe that MINPACK-ESCORT produces output that is identical to MINPACK-C’s output. We also observe that all outputs of MINPACK-FTFP differ from MINPACK-C. Specifically, 321 values differ between the outputs of MINPACK-FTFP and MINPACK-C. We ana-

⁹<https://github.com/devernay/cminpack>

Function	Min.	Median	Max.
add	16	1,743,272	210,125,824
sub	1,312	6,026,976	84,089,503,744
mul	317	8,587,410	112,134,679,849
div	829	5,834,095	30,899,033,427
sqrt	562	2,815,331	21,257,836,468
floor	0	0	0
ceil	0	0	0
log	1,698	5,908,547	2,705,277,8104
log2	262	5,812,840	13,890,632,367
log10	981	10,105,199	40,631,590,323
exp	132	1,409,624	6,066,894
sin	1,316	4,173,786	40,138,955,131
cos	2,166	2,241,360	10,127,702
tan	717	5,576,540	40,126,401,802
pow	522	3,425,870	26,876,068,127
fabs	352	3,129,984	40,134,770,688

Table 7: Floating-point difference for 10,000 operations on random inputs in terms of Unit of Least Precision (ULP) in **FTFP** versus **Musl C library**. Since we observe zero ULP distance between Escort’s results and Musl’s results, this table omits Escort’s results.

< 10 ⁻⁵	10 ⁻⁵ to 10 ⁻³	10 ⁻³ to 10 ⁰	10 ⁰ to 10 ³	> 10 ³
49%	9%	21%	10%	11%

Table 8: Distribution of differences in answers produced by MINPACK-FTFP and MINPACK-C. In all, 321 values differ between the outputs of the two programs.

lyze all 321 differences between MINPACK-FTFP and MINPACK-C by classifying them into the following five categories: (1) smaller than 10^{-5} , (2) between 10^{-5} and 10^{-3} , (3) between 10^{-3} and 10^0 , (4) between 10^0 and 10^3 , and (5) larger than 10^3 . As seen in Table 8, almost half of the differences (49%) are extremely small (less than 10^{-5}), possibly arising from relatively small differences between fixed-point and floating-point calculations. However, we hypothesize that differences amplify from propagation, since nearly 42% of the differences are larger than 10^{-3} .

7 Performance Evaluation

We now evaluate the end-to-end application performance impact of Escort’s floating-point library and Escort’s control flow obfuscation.

Application	Escort Overhead	Static (LLVM) Floating-Point Instruction Count
433.milc	29.33×	2,791
444.namd	57.32×	9,647
447.dealII	20.31×	21,963
450.soplex	4.74×	4,177
453.povray	82.53×	25,671
470.lbm	56.19×	711
480.sphinx3	52.46×	629
MEAN	32.63× (geo. mean)	9,370 (arith. mean)

Table 9: Overhead of SPEC-ESCORT (SPECfp2006 using Escort operations) relative to SPEC-LIBC (SPECfp2006 using libc).

7.1 Impact of Floating-Point Library

This section evaluates the performance impact of Escort on the SPEC floating point benchmarks, as well as on a security-sensitive program $\text{SVM}^{\text{light}}$, a machine-learning classifier.

Evaluation Using SPEC Benchmarks. We use the C and C++ floating-point applications in the SPEC CPU 2006 benchmark suite with reference inputs. We generate two versions of each program—the first version (SPEC-LIBC) uses the standard C library functions, and the second version (SPEC-ESCORT) uses functions from the Escort library¹⁰. We compile the SPEC-LIBC program using the Clang/LLVM 3.8 compiler with the -O3 flag, and we disable auto-vectorization while compiling the SPEC-ESCORT program. The following results demonstrate the *worst* case performance overhead of Escort for these programs, since we transform *all* floating-point operations in SPEC-ESCORT to use the Escort library. More precisely, we do not reduce the number of transformations either using taint tracking or using SMT solvers.

Table 9 shows that Escort’s overhead is substantial, with a geometric mean of 32.6×. We expect a lower average overhead for applications that use secret data, since taint tracking would reduce the number of floating-point operations that would need to be transformed.

Evaluation Using $\text{SVM}^{\text{light}}$. To evaluate Escort’s overhead on a security-sensitive benchmark, we measure Escort’s performance on $\text{SVM}^{\text{light}}$, an implemen-

¹⁰We also ran the same programs using the FTFP library, but the programs either crashed due to errors or ran for longer than two hours, after which they were manually terminated.

Test Case	Overhead for Training	Overhead for Classification
#1	8.66×	1.34×
#2	30.24×	0.96×
#3	1.41×	1.11×
#4	12.75×	0.92×
GEO MEAN	8.28×	1.07×

Table 10: Overhead of Escort on $\text{SVM}^{\text{light}}$ program.

tation of Support Vector Machines in C, using the four example test cases documented on the $\text{SVM}^{\text{light}}$ website¹¹. We mark the training data and the classification data as secret. Before replacing floating-point computations, Escort’s taint analysis discovers all floating-point computations that depend on the secret data, thus reducing the list of replacements. We also instruct Escort to query the Z3 SMT solver to determine whether candidate floating-point computations could use subnormal operands. Escort then replaces these computations with secure operations from its library. We compile the baseline (non-secure) program using the Clang/LLVM 3.8 compiler with the -O3 flag, and we disable auto-vectorization while compiling $\text{SVM}^{\text{light}}$ with Escort. We measure the total execution time using the RDTSC instruction. Table 10 shows that Escort’s overhead on $\text{SVM}^{\text{light}}$. We observe that Escort’s overhead on $\text{SVM}^{\text{light}}$ is substantially lower than that on SPEC benchmarks. Using the md5sum program, we verify that the output files before and after transformation of $\text{SVM}^{\text{light}}$ are identical.

7.2 Impact of Control Flow Obfuscation

To compare the performance impact of Escort’s control flow obfuscation technique with that of Raccoon, we use the same benchmarks that were used to evaluate Raccoon [28], while compiling the baseline (non-transformed) application with the -O3 optimization flag. Although both Escort and Raccoon obfuscate control flow *and* data accesses, we compare the cost of control flow obfuscation only, since both Escort and Raccoon obfuscate data accesses using the identical technique. Table 11 shows the results.

We find that programs compiled with Escort have a significantly lower overhead than those compiled with Raccoon. Escort’s geometric mean overhead is 32%, while that of Raccoon is 5.32×. The worst-case overhead for Escort is 2.4× (for ip-tree).

The main reason for the vast difference in overhead is that Raccoon obfuscates branch instructions at *execution* time, which requires the copying and restoring of

¹¹<http://svmlight.joachims.org/>

Benchmark	Raccoon Overhead	Escort Overhead
ip-tree	1.01×	2.40×
matrix-mul	1.01×	1.01×
radix-sort	1.01×	1.06×
findmax	1.01×	1.27×
crc32	1.02×	1.00×
genetic-algo	1.03×	1.03×
heap-add	1.03×	1.27×
med-risks	1.76×	1.99×
histogram	1.76×	2.26×
map	2.04×	1.01×
bin-search	11.85×	1.01×
heap-pop	45.40×	1.44×
classifier	53.29×	1.24×
tax	444.36×	1.67×
dijkstra	859.65×	1.10×
GEO MEAN	5.32×	1.32×

Table 11: Performance comparison of benchmarks compiled using Raccoon and Escort. We only compare the control flow obfuscation overhead, since both Raccoon and Escort use the same technique for data access obfuscation.

the stack for each branch instruction. Since the stack can be arbitrarily large, such copying and restoring adds substantial overhead to the running time of the program. On the other hand, Escort’s code rewriting technique obfuscates code at *compile* time using basic block predicates, which enables significant performance boosts on the above benchmarks.

8 Conclusions

In this paper, we have presented Escort, a compiler-based tool that closes side channels that stem from floating-point operations. Escort prevents an attacker from inferring secret floating-point operands through the timing channel, though micro-architectural state, and also through off-chip digital side channels, such as memory address trace.

Escort uses native SSE instructions to provide speed and precision. Escort’s compiler-based approach enables it to support a significantly larger number of floating-point operations (112) than FTFP (19).

Escort’s design motivates further research into hardware support for side-channel resistant systems. For example, by allowing software to control the timing of integer instruction latencies and their pipelined execution, Escort’s guarantees could be extended to instructions beyond floating-point instructions.

Acknowledgments. We thank our shepherd Stephen McCamant and the anonymous reviewers for their helpful feedback. We also thank David Kohlbrenner for giving us the Firefox timing attack code. We are grateful to Jia Chen for providing us the pointer analysis library, and to Joshua Eversmann for help with code and discussions. This research was funded in part by NSF Grants DRL-1441009, CNS-1314709, and CCF-1453806, CFAR (one of the six SRC STARnet Centers sponsored by MARCO and DARPA), and a gift from Qualcomm.

References

- [1] ABADI, M., ET AL. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Computing Research Repository abs/1603.04467* (2016).
- [2] ACİIÇMEZ, O., KOÇ, Ç. K., AND SEIFERT, J.-P. On the Power of Simple Branch Prediction Analysis. In *Symposium on Information, Computer and Communications Security* (2007), pp. 312–320.
- [3] ANDRYSCO, M., KOHLBRENNER, D., MOWERY, K., JHALA, R., LERNER, S., AND SHACHAM, H. On Subnormal Floating Point and Abnormal Timing. In *Symposium on Security and Privacy (S&P)* (2015), pp. 623–639.
- [4] BRUMLEY, D., AND BONEH, D. Remote Timing Attacks are Practical. *Computer Networks* 48, 5 (2005), 701–716.
- [5] CLEEMPUT, J. V., COPPENS, B., AND DE SUTTER, B. Compiler Mitigations for Time Attacks on Modern x86 Processors. *Transactions on Architecture and Code Optimization* 8, 4 (Jan. 2012), 23:1–23:20.
- [6] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Network and Distributed System Security Symposium* (2015).
- [7] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), pp. 337–340.
- [8] DEMMEL, J. W. Effects of Underflow on Solving Linear Systems. Tech. Rep. UCB/CSD-83-128, EECS Department, University of California, Berkeley, Aug 1983.
- [9] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic Analysis: Concrete Results. In *Third International Workshop on Cryptographic Hardware and Embedded Systems* (2001), pp. 251–261.
- [10] GROSSSCHÄDL, J., OSWALD, E., PAGE, D., AND TUNSTALL, M. Side-Channel Analysis of Cryptographic Software via Early-terminating Multiplications. In *International Conference on Information Security and Cryptology* (2010), pp. 176–192.
- [11] ISLAM, M. S., KUZU, M., AND KANTARIOGLU, M. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Network and Distributed System Security Symposium, NDSS* (2012).
- [12] JANA, S., AND SHMATIKOV, V. Memento: Learning Secrets from Process Footprints. In *Symposium on Security and Privacy (S&P)* (2012), pp. 143–157.
- [13] KAHAN, W. Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard. *Interval Mathematics* (1980), 99–128.
- [14] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology* (1996), pp. 104–113.

- [15] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential Power Analysis. In *19th Annual International Cryptology Conference on Advances in Cryptology* (1999), pp. 388–397.
- [16] KONG, J., ACIÇMEZ, O., SEIFERT, J., AND ZHOU, H. Hardware-Software Integrated Approaches to Defend Against Software Cache-Based Side Channel Attacks. In *International Conference on High-Performance Computer Architecture* (2009), pp. 393–404.
- [17] LATTNER, C., AND ADVE, V. S. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization* (2004), pp. 75–88.
- [18] LIU, C., HARRIS, A., MAAS, M., HICKS, M., TIWARI, M., AND SHI, E. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), pp. 87–101.
- [19] LUK, C., ET AL. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation* (2005), pp. 190–200.
- [20] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Conference on Computer and Communications Security* (2013), pp. 311–324.
- [21] MARTIN, R., DEMME, J., AND SETHUMADHAVAN, S. Time-Warp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks. In *International Symposium on Computer Architecture* (2012), pp. 118–129.
- [22] MASTI, R. J., ET AL. Thermal Covert Channels on Multi-core Platforms. In *USENIX Security Symposium* (2015), pp. 865–880.
- [23] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *International Conference on Information Security and Cryptology* (2005), pp. 156–168.
- [24] MUCHNICK, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.
- [25] MULLER, J.-M. On the definition of $\text{ulp}(x)$. Tech. Rep. 2005-009, ENS Lyon, February 2005.
- [26] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *RSA Conference on Topics in Cryptology* (2006), pp. 1–20.
- [27] PERCIVAL, C. Cache Missing for Fun and Profit. In *Proceedings of the Technical BSD Conference* (2005).
- [28] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *USENIX Conference on Security Symposium* (2015), pp. 431–446.
- [29] REN, L., YU, X., FLETCHER, C., VAN DIJK, M., AND DEVADAS, S. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In *International Symposium on Computer Architecture* (2013), pp. 571–582.
- [30] SAKURAI, K., AND TAKAGI, T. A Reject Timing Attack on an IND-CCA2 Public-key Cryptosystem. In *International Conference on Information Security and Cryptology* (2003), pp. 359–374.
- [31] SCHINDLER, W. A Timing Attack Against RSA with the Chinese Remainder Theorem. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2000), pp. 109–124.
- [32] SHI, E., CHAN, T. H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *Advances in Cryptology* (2011), pp. 197–214.
- [33] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C. W., REN, L., YU, X., AND DEVADAS, S. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Conference on Computer and Communications Security* (2013), pp. 299–310.
- [34] TUKEY, J. *Exploratory Data Analysis*. Pearson, 1977.
- [35] WANG, Y., FERRAIUOLO, A., AND SUH, G. E. Timing Channel Protection for a Shared Memory Controller. In *International Symposium on High Performance Computer Architecture* (2014), pp. 225–236.
- [36] WANG, Z., AND LEE, R. B. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture* (2007), pp. 494–505.
- [37] WANG, Z., AND LEE, R. B. A Novel Cache Architecture with Enhanced Performance and Security. In *International Symposium on Microarchitecture* (2008), pp. 83–93.
- [38] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Predictive Mitigation of Timing Channels in Interactive Systems. In *Conference on Computer and Communications Security* (2011), pp. 563–574.
- [39] ZHANG, Y., AND REITER, M. K. Doppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Conference on Computer and Communications Security* (2013), pp. 827–838.

ÜBERSPARK[†]: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor

Amit Vasudevan*, Sagar Chaki**, Petros Maniatis***, Limin Jia**** and Anupam Datta****

* amitvasudevan@acm.org – CyLab/Carnegie Mellon University

** chaki@sei.cmu.edu – SEI/Carnegie Mellon University

*** maniatis@google.com – Google Inc.

**** {liminjia,danupam}@cmu.edu – CS/ECE Carnegie Mellon University

Abstract—We present überSpark (üSpark), an innovative architecture for compositional verification of security properties of extensible hypervisors written in C and Assembly. üSpark comprises two key ideas: (i) endowing low-level system software with abstractions found in higher-level languages (e.g., objects, interfaces, function-call semantics for implementations of interfaces, access control on interfaces, concurrency and serialization), enforced using a combination of commodity hardware mechanisms and lightweight static analysis; and (ii) interfacing with platform hardware by programming in Assembly using an idiomatic style (called CASM) that is verifiable via tools aimed at C, while retaining its performance and low-level access to hardware. After verification, the C code is compiled using a certified compiler while the CASM code is translated into its corresponding Assembly instructions. Collectively, these innovations enable compositional verification of security invariants without sacrificing performance. We validate üSpark by building and verifying security invariants of an existing open-source commodity x86 micro-hypervisor and several of its extensions, and demonstrating only minor performance overhead with low verification costs.

1. INTRODUCTION

The modern hypervisor stack is, by necessity, extensible. Hypervisors not only enable the old-hat style of customization, such as modularity for device drivers, but are further extended with convenient functionality for security services such as attestation, debugging, tracing, application-level integrity and confidentiality, trustworthy resource accounting, on-demand I/O isolation, trusted path, and authorization [14], [18], [22], [49], [53], [57], [62], [64], [65], [71], [74], [75], [77], [80], [83]–[86]. Further, the overwhelming majority of the deployed hypervisor codebase is written in low-level C and Assembly, due to hardware accesses, developer familiarity, and performance requirements.

1.1. Problem – The unbridled growth of these extensible hypervisors, while enabling useful functionality,

raises significant security concerns. As the size and complexity of these systems increase – not to mention the number of extensions, which may be active in arbitrary combinations – so has the incidence of security-related bugs. Indeed exploitable bugs in extension interfaces have led to compromises in various hypervisors ranging from complex VMMs to micro-hypervisors [2], [3], [26], [27], [44]. Thus, higher assurance in the security properties offered by hypervisors is critically important.

1.2. Solution – We address this challenge by developing überSpark (üSpark), an architecture for building extensible hypervisors that: (a) is *compatible with commodity systems*; (b) enables *automated compositional verification of security properties*; and (c) produces *performant systems*. Compatibility with commodity systems is crucial to impacting developers and deployment ecosystems. üSpark supports development and verification directly at the C and Assembly source and enables access to more commodity hardware features. It is thus distinct from prior approaches that sacrifice commodity compatibility by employing new programming languages or hardware models [33], [36], [81]. Compositionality means that extensible systems can be verified modularly, rapidly, and independently as they are implemented. Specifically, when an extension is added, üSpark does not require complete system re-verification to re-establish properties. While this goal guides much work in high-level languages, achieving it for low-level languages is a significant challenge. Furthermore, it distinguishes us from verification of full functional correctness [31], [33], [43]. We focus only on security invariants – memory separation, control-flow integrity, information flow – and other extension properties that can be formulated as invariants. We verify such properties directly, compositionally, and automatically on the C and Assembly implementation. This helps bring to commodity-compatible hypervisors those on-going approaches, which offer full functional correctness, but we also enable precise reasoning on untrusted and unverified system code. Finally, the üSpark hypervisor’s performance is close to that of a commodity unverified system.

[†]In the fictional Transformers universe, the AllSpark is a powerful object capable of creating a new Transformer by bestowing ordinary machinery with sparks – the building blocks of a Transformer. In a similar vein, ÜBERSPARK bestows ordinary hypervisors with verifiable objects (ÜOBJECT) for automated compositional security analysis.

Key to the power of üSpark is the enforcement of verifiable-object abstractions to hypervisors. The basic building block is a *üobject*, which encapsulates specific system resources and provides an interface for accessing them – with a well-defined behavioral contract comprising a use manifest along with formal behavior specifications. A üobject may represent core components of a hypervisor or an extension and may be concurrent or sequential. Public methods of concurrent üobjects are invoked in parallel by multiple cores whereas sequential üobjects are implemented as monitors, guarding all method invocations via a per-üobject lock. üObjects communicate with each other via function calls.

There are two special üobjects: *prime* sets up a sane initial state, while *sentinel* ensures control-flow semantics even when üobjects with different levels of privilege and trust invoke each other. Together, they enable compositional inductive proofs of security properties expressed as invariants over sequential üobjects via source code analysis and hardware assumptions [8]. A third group of special üAPI üobjects allow access to shared resources enabling state-of-the-art tools for automatic verification of sequential C code to be soundly applied to verifying security properties, while still allowing multi-threaded high-performance applications.

In keeping with our first and second design goals, üSpark enforces verifiable-object abstractions using a combination of commodity hardware mechanisms (page-tables and de-privileging) and light-weight static analysis, leveraging off-the-shelf C99 source-code analysis and certified-compilation tools. üObjects, including prime and sentinel, are automatically and modularly verified using Frama-C [41], an industrial-strength software analysis and verification framework. We use standard and custom Frama-C plug-ins to perform static verification checks that include: per-üobject behavioral contracts (via a standard weakest-precondition plug-in); abstract variable assertions that enable behavioral asserts as well as üobject control-flow integrity (via a standard abstract-interpretation plug-in on stack frames and other variables); syntactic checks that ensure conformance with a restricted C99 syntax and logical de-privileging of üobjects (via a standard abstract syntax tree analysis plug-in); and, composition checks that enable client üobjects that share a common server üobject to compose soundly (via a custom composition-check plug-in).

üSpark also provides an idiomatic use of Assembly, called CASM, to separate it from C code during system construction. During analysis with Frama-C, the CASM code is replaced by a C99 hardware model which models key commodity hardware features. Our custom Frama-C plug-in checks that the syntactic restrictions imposed by CASM are respected by every üobject. The verified üobjects are then compiled into executable binaries. Dur-

ing üobject compilation, all C99 code is processed using the certified CompCert compiler [12] while each CASM instruction is replaced by the corresponding Assembly instruction by our custom Frama-C plugin. The CASM language is designed to ensure that the C and Assembly code operate on disjoint state. Our longer-term goal is to guarantee that the verified source code properties carry over to the binary by leveraging the C-Assembly separation and cleanly extending the bisimulation proof of the CompCert compiler to encompass hardware state and Assembly code. In addition, we aim to ensure the semantic equivalence between the hardware model and the corresponding Assembly instructions. Proving these guarantees formally appears straightforward, and need only be done once for the üSpark framework, but we leave it to future work.

The üSpark object abstraction is distinguished from other systems in that it allows many fine-grained objects in privileged mode. Static analysis enforces logical deprivileging of those objects – e.g., a hypervisor module running in host-mode ring 0 is precluded from accessing page-table structures, thereby being “logically” deprivileged – while control transfer between them does not involve a context switch, thereby significantly helping with system performance, our third design goal.

1.3. Contributions – (a) We present üSpark, an innovative architecture providing verifiable object abstractions for automated compositional verification of hypervisor security properties while targeting commodity compatibility and performance (§4,§5). (b) We use üSpark to incrementally develop and verify security properties of an existing open-source commodity x86 micro-hypervisor with multiple independent security extensions (hypervisor and extensions realized as 11 üobjects with 7001 SLoC; 5544 and 2079 lines of annotations and hardware model; §6,§7). (c) We carry out a comprehensive evaluation showcasing verification metrics, development effort and performance, and report on our experience (1 person yr; üobject verification times from 1–23 minutes with a cumulative time \approx 1hr; 2% average runtime overhead over native micro-hypervisor applications with guest performance unaffected; §8,§9).

2. A MOTIVATING EXAMPLE

To motivate and explain üSpark, we use as a running example, a hypervisor that closely corresponds to our case study. Imagine the hypervisor managing a multi-CPU guest, and supporting optional security extensions that implement various guest-specific and system-wide security properties. The hypervisor manages system devices used by itself, by extensions, and by the guest. System devices execute device firmware in parallel with the CPUs and perform DMA. The hypervisor and extensions are written in C and Assembly.

The hypervisor leverages CPU capabilities, such as memory-mapped I/O (MMIO) and legacy I/O, for system-to-device interaction; it initializes boot CPU (BSP) state; it sets up memory page tables, as well as device allocations and DMA protections (e.g., via an IOMMU); it initializes multi-CPU support via the Local Advanced Programmable Interrupt Controller (LAPIC) and activates other CPUs and sets up their memory page tables and appropriate protections. Constructing a verified hypervisor of this sort, the developers must not only build it and test it well, but also verify its code against a set of general safety properties (e.g., memory integrity) as well as functional invariants on hardware and software state (e.g., IOMMU, LAPIC, CPU states).

Consider now adding two new *verified* extensions to the hypervisor: **hyperdep**, which ensures that guest VM data pages are non-executable; and (b) **sysclog**, which ensures that every system call issued by the guest is logged via a dedicated network card to an external trusted entity on the network. In order to preserve the verified status of the system, the developers must prove that: (a) memory integrity is not violated by the extensions; (b) each extension provides its claimed property to guests configured to use it; and (c) the extensions are used in tandem by a guest if and only if they provide a well-defined compositional property (e.g., separability). This is non-trivial, since it requires the construction and verification of inductive invariants that imply the core security properties of the hypervisor, and those of enabled extensions. Also, since extensions are optional, verification must account for all possible configurations – e.g., enabling either **hyperdep**, or **sysclog**, or both – while avoiding the combinatorial blowup.

Of course, history tells us that two extensions are never enough for any extensible system. What is more, not all extensions come from the same developers or with the same pedigree. Consider, for instance, an *unverified*, strictly optional extension to the hypervisor; this might be an extension that provides essential functionality, but has not been verified, and is taken as an acceptable risk. For our example, let us use **aprveexec**, an extension that ensures that guest code pages contain only read-only, whitelisted content. As with **hyperdep** and **sysclog**, core hypervisor properties, and the properties of other extensions should not be violated by running **aprveexec**, and the risk of running **aprveexec** should only be suffered by a guest that explicitly enables it and relies on its presumed properties. Note that the guest itself, unless it is verified as rigorously as the rest of the hypervisor, is such an unverified component in the system.

3. GOALS AND ASSUMPTIONS

3.1. Goals – Our overarching goal is to enable development of performant extensible hypervisors offering pro-

ofs of wide-ranging properties on their code, including low-level memory safety, control-flow guarantees, and information flow, as well as higher-level properties such as trusted network logging (**sysclog**) and data execution prevention (**hyperdep**), going all the way up to security properties spanning both hardware and software states (IOMMU, LAPIC, network-card and CPU). Also, verification must support properties over shared system states: e.g., both **hyperdep** and **sysclog** manipulate guest memory protections via the same guest page-tables. Our design goals fall broadly in three categories.

3.1.1. Compositionality: When new components are added, or existing components changed, human re-verification effort should be limited to the changed codebase, yet it should provide guarantees about the entire system under all possible configurations.

3.1.2. Legacy Compatibility & Usability: Our development and verification approach must integrate into the existing hypervisor C and Assembly language programming ecosystem, and cover the entire source code base including commodity hardware and guest OS. We must support extensions that are unverified in order to preserve the legacy ecosystem. However, unverified code (e.g., the guest) must not violate system properties established by verified code. Our development and verification techniques must foster wider adoption by hypervisor developers. We envision that entry-level developers will rely on basic building blocks to provide simple properties while seasoned developers will harness the full verification power to provide stronger guarantees.

3.1.3. Performance: Verification must not preclude aggressive code optimizations for individual components, including extensions, and must not adversely affect runtime performance. Further, commodity guest OS on multi-core hardware must be supported.

3.2. Non-goals – We do not aim for full functional correctness (i.e., verifying that the implementation behaves exactly as specified in a high-level abstraction). This separates the concerns of showing how a complex low-level system achieves low-level formal properties from how those low-level properties refine a high-level abstract model; we focus on the former, since it is a hard and as yet open problem, whereas much on-going work tackles the latter [31], [42].

3.3. Attacker Model and Assumptions – We assume that the attacker does not have physical access to the CPU, memory, chipset or other verified extension-specific system devices (our hardware TCB). Other system devices, the guest OS, and unverified extensions are under the attacker’s control. This is reasonable since a majority of today’s attacks are mounted by malicious software or untrusted system devices. We assume that our hardware TCB is functionally correct, and we have load-time integrity, i.e., the verified hypervisor is the one

securely loaded onto the hardware at boot time. Finally, we assume that the verification tools we use are sound.

4. ÜSPARK ARCHITECTURE

We next describe our architecture, and how it addresses our goals (§3.1) via verifiable object abstractions (Fig 1)

4.1. üObjects – The basic building block in üSpark– the “üobject” – is used to contain any system component including verified and unverified hypervisor and guest blobs and system devices. Logically, a üobject is a singleton object guarding some otherwise indivisible resources (e.g., registers, memory, devices) and implementing public methods to access them. Public methods are essentially regular function signatures but can be restricted to specific callers (§4.2.1). Every üobject also has a special public method, *init*, to set up the üobject in a known-good initial state. A üobject may be concurrent or sequential. The public methods of a concurrent üobject can be invoked in parallel on multiple cores. In contrast, at most one core can invoke the methods of a sequential üobject at a time, as with a traditional monitor. When multiple cores are active, sequential execution is enforced via per-üobject locks.

Each üobject defines its functionality using C and Assembly. Assembly language for a verified üobject is written using CASM, a dialect of C in which Assembly instructions are encoded within regular C functions (CASM functions) via C-like pseudo-function calls (CASM instructions¹). For example, for the x86 instruction `movcr3` involving register `eax` there is a corresponding CASM pseudo-function called `ci_movl_eax_cr3`. Each CASM instruction pseudo-function is defined in the üSpark hardware model (§7.1.2) and bridges the shift between the reference C semantics and the hardware instructions (e.g. access to memory and to registers). During verification, each CASM instruction is replaced by the C source code from the hardware model. The resulting C-only program is verified for required properties. CASM functions are verified to respect the C application binary interface (ABI), which is crucial for the soundness of verification. During compilation, all C functions are processed via a certified compiler while each CASM instruction is replaced by the corresponding Assembly instruction. In contrast to prior code-level verification approaches (§10), CASM supports two-way nested C to Assembly calling with full device modeling. This allows using various verification techniques to prove (higher-level) properties on device states other than just memory and numeric

safety (§7.2). CASM also allows aggressive compiler optimizations of the callee C functions including inlining as per compiler specifications, resulting in optimal runtime performance (§8.3). We envision further optimizations including inlining of hand-written CASM code as part of our future work (§11.2).

Beyond defining its own functionality, a üobject is also accompanied by a behavior *contract*. This consists of a *use manifest* (§4.3) and a formal *behavior specification* of its own public interface, which guarantee that if a certain assumption is satisfied in how a public method is invoked, then a property on the return values is guaranteed to hold upon return of that method, without mention of internal üobject state.

Every üobject is held to a number of invariants, which together guarantee its adherence to the verifiable-object abstraction. These invariants include memory and (internal) control-flow integrity, so that the code can be reasoned about; and satisfaction of the formal contract, so that the contract alone may overapproximate the üobject, thereby enabling compositional verification; as well as correct initialization. The invariants are discharged via assumptions on the hardware and proofs on the source code of the üobject, and on the contract of üobjects it interacts with (§5, §7.2).

While our use of object encapsulation is similar to existing micro-kernel architectures [42] and prior capability systems [32], [63], üSpark is distinguished by privileged disaggregation, i.e., multiple verified privileged üobjects can be logically deprivileged. This enables us to achieve the sweet spot with both high performance (there is no hardware de-privileging overhead; §8.3.1) and compositional verification (privileged üobjects can be verified separately; §7.2).

4.1.1. Prime: is the first üobject to execute in a üSpark enabled hypervisor. Prime is verified to satisfy its contract which is: to set up the required system interfaces and associated policies, establish operating stacks, prepare the platform CPU cores, invoke the *init* methods of other üobjects to initialize their state, and kick-start üobject interactions.

4.2. üObject Interaction – A üobject interacts with another by invoking a public method in its interface with appropriate parameters. All verified üobjects operate on a single stack (one per CPU core) that is set up initially by the prime. Each unverified üobject uses its own, separate stack. The verifiable-object abstraction requires üobject-to-üobject control-flow integrity (otherwise returns could land at arbitrary üobject program sites, access controls would be violated, etc.). Therefore, üobjects must also be verified to use their stack correctly (another invariant). For unverified üobjects, that also means that stacks must be switched to/from the unverified üobject stack and a separate shadow stack must be maintained for storing

¹CASM syntax is similar to existing “asm” keywords supported by traditional C compilers for integrating Assembly language instructions. However, CASM provides a more principled way to integrate Assembly instructions tailored for verification while retaining performance.

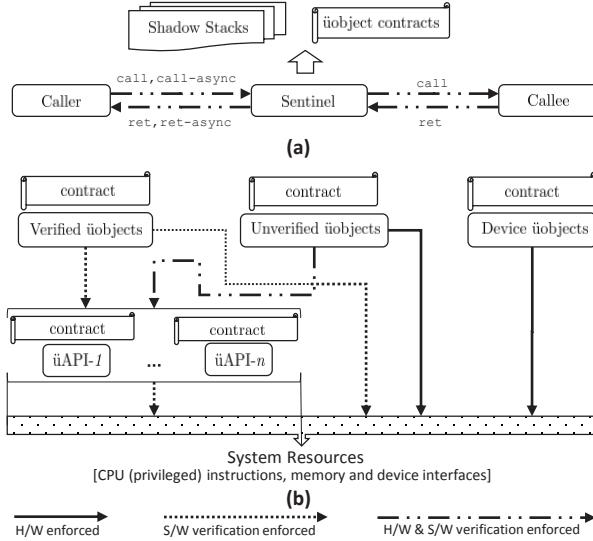


Fig. 1: überSpark: enforces verifiable object abstractions using a combination of commodity hardware and software verification mechanisms to: (a) translate synchronous (call) and asynchronous (e.g., exceptions, intercepts) inter-üobject control transfers, to establish pure function call-return semantics; and (b) establish üobject resource confinement.

return addresses during control transfers. The special *sentinel* üobject performs (verifiably) this functionality.

4.2.1. Sentinel: is a special üobject that mediates interactions among other üobjects. Thus, an invocation of a public method of a callee üobject by a caller üobject is intercepted by the sentinel and dispatched only after a number of optional runtime checks have succeeded.

These runtime checks logically ensure that the caller may invoke a given public method on the callee according to the üobject manifest (§4.3). For example, an extension can be split between a top half and a bottom half as with traditional device drivers (in our case study, **sysclog** could shed its networking code into a separate üobject, **sysclognw**, that only takes transmission requests from **sysclog**, and is the only authorized user of a separate NIC dedicated to logging), ensuring that only the top half may invoke the bottom half at runtime, while still keeping the two isolated from each other and independently verifiable. If caller and callee are both verified, then no runtime check is required, since static analysis enforces the call policy (§4.3). If one is unverified, the sentinel consults the policy dynamically and allows or rejects the call accordingly.

Besides the runtime checks, the sentinel is responsible for transferring control among üobjects. If both are verified, the control transfer is just a function call. But if either is unverified, the sentinel must employ the appropriate control-transfer method for the isolation mechanism imposed on the unverified üobject (e.g., if using ring-based isolation, switch privilege levels and stacks, marshal arguments, etc.). The sentinel may implement control transfers according to a number of concrete

ways (hardware virtual machines, software fault isolation, etc.), while still adhering to the high-level invariant for isolation. For example, in our micro-hypervisor implementation, the sentinel traverses both ring-based isolated üobjects, and hardware virtual machines (§6).

The sentinel is an üobject, so it adheres to the same invariants as regular üobjects, but it is also verified to implement its function correctly (perform the checks, properly transfer control, etc.).

4.3. üObject Resource Confinement – üSpark implements *iobject resource confinement* in which distinct system resources are: (a) managed by designated üobjects, (b) protected from access by unauthorized üobjects, and (c) regulated in their use by authorized client üobjects. Such resources include üobject local memory (code, data, stack), system memory (e.g., BIOS data, free memory), CPU state and privileged instructions, system devices and I/O regions. Every üobject includes a *use manifest* in its contract that describes which resources it may access. It is held to the property that it can only use the resources declared in its manifest.

For verified üobjects, üSpark employs a hardware model identifying CPU interfaces to system resources (e.g., I/O and designated memory instructions interface to system devices, instructions that can modify CPU model specific register states, etc.) and static analysis to ensure that access to those interfaces respects the üobject's manifest (§7). For example, **sysclog**'s manifest shows that it may access the dedicated NIC for its remote logging, and static analysis ensures that the code for **sysclog** may access only that NIC, nor can any other üobject access **sysclog**'s NIC.

In contrast, unverified üobjects are held to their use manifests via more direct enforcement mechanisms, such as hardware MMU and privilege protections (virtualization, de-privileging) and software manipulations (e.g., SFI). Unverified üobjects can also be granted direct access to exclusively held system devices so they can perform I/O without any performance overhead (e.g., a guest OS üobject is allocated all the devices except the LAPIC and **sysclog**'s network card). Device üobjects use DMA as their interface to other üobjects. üSpark uses hardware IOMMU capabilities to ensure that device üobjects are restricted to perform DMA only to designated üobject DMA memory regions.

4.3.1. üAPI üobjects: are a special set of üobjects that encapsulate shared resources over which system properties are established (§6.4). For example, guest OS üobject memory and CPU state are manipulated by multiple extensions (**hyperdep** and **sysclog**). üSpark enforces a composition check (§7.2.1), which for a given set of üAPI üobjects checks if a set of “client” üobjects are composable. Note that every üAPI üobject also performs compositability checks at runtime for invocations from

unverified üobjects. Such composability checks reason about the use-manifest portion of a client üobject’s contract, which constrains how that üobject invokes the üAPI’s public methods, ensuring some system-specific and üAPI-specific composability guarantee, such as separability. Client üobjects must satisfy the property that whenever they invoke a üAPI call, they obey their own use manifest, and üSpark discharges this property via static analysis on verified üobjects or runtime sentinel checks for unverified üobjects.

4.4. üSpark Blueprint – üSpark also defines a hypervisor *blueprint* (üBP), which a hypervisor implementation is held to. The üBP is a high-level control-flow graph that divides hypervisor execution into three phases: startup, intercept, and exception handling which can in turn be customized based on the actual number of system üobjects and their interactions (Figure 2; §6). The üBP along with our high-level proofs (§5) enables us to abstract the hypervisor, running on multi-core platform hardware with system devices and DMA, as a non-deterministic sequential program. This, in turn, allows us to prove invariant properties of üobjects, and the hypervisor as a whole, via sequential source-code verification. Further, the üBP also enforces that fragile bits of the hardware state (e.g., CPU and IOMMU) are only touched within a monitor. This, allows us to prove invariant properties encompassing hardware states and keeps our hardware model simple by precluding modeling of concurrent hardware accesses (§7.1.2).

5. ÜSPARK FORMALISM

We present a formalization of üSpark that justifies the soundness of our analysis. For brevity, we first give an overview of the formal reasoning followed by our high-level verification approach and related theorems. Full proof details can be found in our technical report [73].

5.1. üSpark Formalism Overview – üSpark reasoning relies foundationally on a set of invariants – properties that must hold throughout the execution of a üSpark hypervisor (Appendix A). The invariants are divided into üSpark system invariants and üSpark general programming invariants (those that pertain specifically to üobject C and CASM functions). Each invariant is proved by reducing it further to a set of *proof-assumptions on hardware* (PAHs) and *proof-obligations on code* (POCs) using the üSpark blueprint (üBP; Fig. 2). POCs are then discharged on all üSpark verified üobjects including the prime and sentinel using specific verification tools and techniques (§7). A hypervisor implementation is compliant with üSpark– and therefore amenable to compositional reasoning – if it satisfies all the üSpark invariants. Full details of invariant-to-PAH/POC mappings, a one-time effort, is described in [73]. At a high level, üSpark invariants ensure the hypervisor implementation follows

the üBP and that prime is correct, and the first to start in the system, and that it sets up memory protections, stacks, and CPUs, before starting other execution contexts in a well-defined state. The remaining invariants guarantee that üobjects have memory and control-flow integrity, and the sentinel properly transfers control among them, respecting the concurrent or sequential designation.

5.2. Verification Approach and Theorems – There are two tasks in verifying properties of a üSpark hypervisor: (a) showing that it obeys the üSpark invariants; and (b) showing that it obeys any hypervisor/extension-specific invariant properties. The benefit of (a) is that developers can express system-specific properties in terms of üobjects and their interactions with each other, yet verify those properties separately on each individual üobject in isolation, and on the ensemble of the behavior contracts of all üobjects, without having to perform slow verification of the combined source code for the whole code base.

Crucial to the model of üobject are CASM programs, defined below. First, we define a *CASM function* as a CompCert-C99 (CC99) function whose body consists only of a block of Assembly instructions that respect the CC99 ABI. A üobject CASM program is a CC99 program such that: (i) all Assembly code appears only in CASM functions; and (ii) these CASM functions preserve the caller C functions’ CPU register state.

Given a üobject CASM program, we are interested in verifying two kinds of properties: (1) invariant properties: whether φ holds at every state (after every instruction), and (2) individual state assertions: whether φ holds at specific program points. We can also specify assumptions (i.e., preconditions), stating that we assume φ holds when a function is called. Verification tools such as Frama-C (§7) take programs annotated with properties to be checked and decide whether the properties hold on all execution traces of the program.

We begin by stating two üSpark theorems essential for the correctness of our approach, which follow directly from the üSpark programming invariants (Appendix A).

Theorem 1 (DISJOINTCASM). *The union of üobject CASM and C functions preserve the existing semantic preservation property of the certified compiler.*

Theorem 2 (EXITSENTINEL). *üobject execution can only exit via the sentinel.*

The next theorem states that each üSpark execution is an interleaving of properly nested executions of üobjects, one on each core (a more formal definition can be found in [73]). Intuitively, it means that üobject calls and returns are properly nested except that the return of an unverified üobject can be an exception, as an unverified

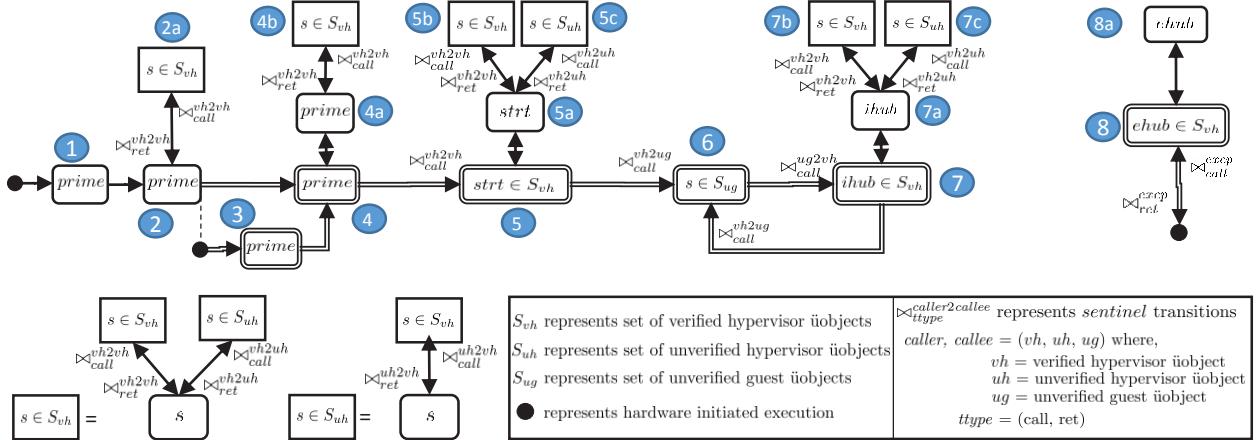


Fig. 2: üSpark Hypervisor Blueprint: startup, intercept and exception handling execution phases. Rounded boxes = üobjects; Square boxes = nested üobject calls; Arrows = intra- and inter-üobject transitions; Single-lines = serialized execution; Double-lines = concurrent execution.

üobject can lie about its return address, but will be caught by the hardware if it steps out of the üobject memory. This theorem enables us to view üSpark semantically as a concurrent object-oriented program, which is then abstracted as a non-deterministic sequential program for verification.

Theorem 3 (NESTEDCALL). Consider a legal execution π of üSpark and a sequential üobject s . The projection of π on executions of s consists of a sequence of properly nested executions of s , each on a specific core.

5.2.1. Hardware Model and Converting Assembly to C: We use C verification tools to verify CASM functions in üobjects by converting Assembly to C. In addition to general-purpose registers (which are preserved to respect the CC99 ABI) these Assembly instructions access special hardware registers (e.g., LAPIC). Let us denote the set of registers accessed by CASM functions in üSpark by \mathcal{R}_{hw} . We introduce a set of fresh C variables (denoted \mathcal{V}_{hw}), one for each register; replace each Assembly instruction accessing \mathcal{R}_{hw} by one or more CC99 statements that operate in a semantically equivalent way over \mathcal{V}_{hw} ; replace each $r \in \mathcal{V}_{hw}$ with v_r in assertions used for specifying hardware state during verification. We refer to the mapping between \mathcal{R}_{hw} and \mathcal{V}_{hw} , and the induced mapping from Assembly instructions to CC99 statements, as our *hardware model*. We assume that this mapping is correct. We refer to the CC99 function obtained by transforming a CASM function f in this manner as \tilde{f} .

5.2.2. Abstract üSpark: We abstract üobjects as a non-deterministic CC99 (NDCC99) program, i.e., a CC99 program with non-deterministic selection of values from finite sets. In particular, the abstract üSpark üBP consists of a set of abstract üobjects, where each abstract üobject \tilde{s} is obtained from the corresponding concrete üobject s by converting each function $g \in p(s)$ to an

abstract function \tilde{g} ; more concretely: by replacing all CASM functions as described above, replacing accesses to data that other cores and devices can modify by non-deterministic values, replacing a call to an unverified üobject by a call to the intercept handler üobject with non-deterministic arguments. The next theorem states that each function g in a sequential üobject refines its abstract version \tilde{g} in that for each properly nested execution of g , there is a corresponding execution of \tilde{g} . This is crucial to the soundness of our verification.

Theorem 4 (EXECREFINE). If g is a function belonging to a sequential üobject such that all Assembly code in g is in a CASM function satisfying all üSpark programming invariants, and c is any core, then for each properly nested execution τ of g on c there is a corresponding execution $\tilde{\tau} \in \llbracket \tilde{g} \rrbracket$ such that: $\tau \equiv \tilde{\tau}$, where $\tau \equiv \tilde{\tau}$ lifts the per-state equivalence to the trace.

We use C verification tools to verify POCs directly on üBP (NDCC99 programs) of üSpark. Theorem 4 allows us to lift the verification results to üobject source-code, formally stated in the following theorem (we only show the statement for invariant properties; the statement for individual state assertions is similar).

Theorem 5 (INVCOMPOSE). Given any sequential üobject s , let \tilde{s} be the üBP abstraction of s . If an invariant property φ holds on every execution of $g(\tilde{s})$, then φ is an invariant property of every execution of s .

6. ÜSPARK HYPERVISOR IMPLEMENTATION

We applied üSpark to XMHF, an open-source micro-hypervisor for the x86 32-bit hardware-virtualized platform [72]. Originally, XMHF consists of a core hypervisor and a single extension (called *hypapp*), that together implement security-specific functionality. The latest version (0.2.2) runs a Ubuntu 12.04 32-bit multi-core guest OS with the core and hypapp at the highest

privilege level and has been used to develop a wide variety of security applications [53], [74], [83], [85], [86]. Our goal is üXMHF – an incrementally developed and verified version with deprivileged components, and multiple hypapps. As a first step, we refactor XMHF into: (a) verified hypervisor (*vh*) üobjects for prime, sentinel, core, üAPIs, and verified hypapps; (b) unverified hypervisor (*uh*) üobjects for unverified hypapps; and (c) unverified guest (*ug*) üobjects for the OS (Figure 2); §8 quantifies this refactoring effort.

6.1. Core, Hypapp and Guest üObjects – We instantiate üXMHF core using three *vh* üobjects: *xcstrt* (startup), *xcihub* (handling *ug* üobject intercepts), and *xcehub* (runtime hardware exception and watchdog handling). We instantiate extensions described in §2 as separate *vh* and *uh* üobjects and add support for multiple hypapps within *xcihub*. Finally, we instantiate a *ug* üobject, *guest* for the guest OS. The *xcstrt* üobject gets control from the prime üobject (§6.2), invokes all registered hypapp üobjects for initialization, and then transfers control to *guest*. The *xcihub* üobject gets control from the sentinel upon any intercept (§6.3) and in turn invokes the hypapp üobjects for *guest* event processing. Upon intercept handling, *xcihub* resumes execution of *guest ug* üobject (Figure 2).

6.2. Prime üObject – The üXMHF boot-loader uses the GETSEC[SENTER] instruction to setup a dynamic root-of-trust and invokes the prime üobject in a hardware protected execution environment with the CPUs in a known good state and interrupts and DMA disabled.

Prime first enumerates devices and uses VT-d IOMMU to restrict their DMA to designated memory regions. It then initializes the *vh* and *uh* PAE page tables and the *ug* 2D EPT page tables for memory protections such that: (i) *vh* page tables map *vh* üobject memory regions, including MMIO, with supervisor privileges, and all *uh* and *ug* üobject memory regions as user with read-write permissions; (ii) each *uh* and *ug* page tables marks only its own region, including MMIO, as user and present; (iii) for *uh* üobjects, all *vh* üobject memory regions are marked supervisor; and (iv) for *ug* üobjects all *vh* and *uh* memory regions including MMIO are marked not-present. Prime uses disjoint CPU I/O bitmaps (which are marked supervisor within *uh* and *ug* üobject page tables) for *uh* and *ug* üobjects’ legacy I/O isolation.

Finally, for each CPU in the system, prime: (a) activates protected-mode with paging and hypervisor-mode via control registers CR0 and CR4 and the VMXON instruction; (b) sets up SYSENTER MSRs, interrupt descriptor table and VM control structure (VMCS) to transfer control to the sentinel; and (c) loads *vh* page tables in CR3 and transfers control to *xcstrt* core startup üobject.

6.3. Sentinel üObject – For *vh* to *vh* üobject control transfers, the sentinel uses an indirect JMP instruction.

The SYSEXIT and SYSENTER fast system call instructions are used *vh* to *uh* control transfers and vice-versa. In such cases, the sentinel loads the *uh* page tables into the CR3 register and transfers control to the *uh* üobject entry point (or return address via the SYEXIT instruction) at the de-privileged level. The sentinel uses the VMLAUNCH instruction for a call from a *vh* to *ug* üobject. It handles intercepts by transferring control to the *vh* *xcihub* üobject and upon return from *xcihub* resumes the *ug* üobject via the VMRESUME instruction. In both cases, it loads the *ug* üobject EPTs prior to the launch. The sentinel handles exceptions by transferring control to the *vh* *xcehub* üobject. Upon return from *xcehub* execution is resumed via the IRET instruction.

6.4. üAPI üObjects – Both the core and hypapp üobjects use üAPI üobjects to influence the *ug* üobject state. This state includes the *ug* üobject EPTs and VMCS. We implement üAPI üobjects *ugmpgtbl* and *ugcpust* which present interfaces to the *ug* üobject EPTs and VMCS respectively. We also implement an additional üAPI üobject *uhcpust* as an interface to shared CPU state between *vh* and *uh* üobjects (e.g., MSRs).

6.5. üObject Runtime Library – üObjects rely on a set of common functionality implemented in the following libraries: (a) *libuc* with memory and string functions; (b) *libucrypt* with SHA-1 functionality; (c) *libustub* with üobject entry and sentinel CASM stubs; and (d) *libuhw* for platform hardware access.

7. ÜSPARK HYPERVISOR VERIFICATION

7.1. Verification and Development Tools – We first describe the verification and development tools we use.

7.1.1. Static Analysis with Frama-C: Frama-C [41] is an industrial-strength C99 static analysis and verification toolkit, written in type-safe OCaml. It has a modular architecture and offers different plugins for distinct styles of analysis. We use the following Frama-C plugins: *Deductive verification* via Frama-C’s Weakest-Precondition (WP) plugin enables the verification of assume-guarantee behavior specifications on C functions. Those specifications are expressed in the Annotated ANSI C Specification Language (ACSL) [25] in terms of the C source variables and operations. The WP plugin verifies such ACSL specifications statically on the body of the function by discharging verification conditions via an ensemble of external SMT solvers. *Abstract interpretation* via Frama-C’s Value plugin analyzes a program using a sound abstraction of its concrete semantics. It is used to prove ACSL assertions placed in the body of the program that express partial specifications about program variables, and can be combined with deductive verification. *Abstract syntax tree (AST) analysis* via Frama-C’s AST plugin performs syntactic analysis on control-flow

graphs and ASTs to enforce syntactic restrictions, e.g., the absence of primitives like function pointers.

7.1.2. Hardware Model: We have implemented a C99 hardware model for the commodity x86 hardware-virtualized platform, by representing platform features such as CPU registers and system-device states as C variables and describing formally how the hardware (should) behave. The hardware model is a re-usable but trusted component. Our hardware model allows for iterative development, modeling only portions of the device used in proving security invariants. This design principle, coupled with serialization enforced by the üSpark architecture blueprint (§4.4), enables us to keep the hardware model simple and amenable to formal validations. Various techniques exist to validate such a hardware model [50], [58] which we plan on exploring as future work (§11).

7.1.3. üSpark Frama-C Plugins: We built üSpark-specific plugins on top of Frama-C as follows: (a) übp – enforces üSpark blueprint; (b) ühwm – embeds hardware model during verification; (c) ücasm – substitutes Assembly mnemonics corresponding to CASM instructions after verification; (d) ücc – enforces general üSpark coding rules; (e) ümf – parses üobject manifest; and (f) ücvf – performs composition check (§7.2.1). These üSpark-specific plugins do not impact the robustness of the Frama-C toolset as we do not modify the kernel or standard plugins. Further, Frama-C’s modular architecture helps us keep üSpark-specific Frama-C plugins small, simple, and amenable to manual audits to ensure correctness (§8.1).

7.1.4. Frama-C and CompCert: In keeping with our longer term goal of guaranteeing that the verified source code properties carry over to the binary, we employ the CompCert [11], [12], [46] certified C99 compiler to compile üobjects. CompCert over-specifies C99 implementation-defined and unspecified behaviors and is formally verified to produce semantically equivalent Assembly from a C99 program. Our choice of Frama-C and CompCert is further justified by their semantic compatibility. We empirically tested Frama-C against CompCert’s C99 specifications and found that both tools had the same treatment of C99 implementation-defined and unspecified behaviors. Further, both tools employ an identical byte-addressable memory model with base addresses and offsets. Therefore, they combine naturally into a powerful analysis and development workflow towards producing verified system binaries.

7.1.5. Soundness Via Weakening: We weaken our execution model in two cases to enable sound reasoning. First, since current state-of-the-art static analyzers including Frama-C largely assume sequential execution, we treat all reads to DMA memory and all memory reads by a concurrent üobject as non-deterministic, for verification

```

1 void ugmpgtbl_setentry(u32 gsid, u32 addr, u64 v) {
2 /*sysclog*/ {v=v&7; v&=~_X; v|= _R; v|=_W;}
3 /*hyperdep*/ {v=v&7; v&=~_X; v|= _R; v|=_W;}
4 /*@assert sysclog: (! (v&_X) && (v&_R) && (v&_W));*/
5 /*@assert hyperdep: (! (v&_X) && (v&_R) && (v&_W));*/
6 }

(a)
7 void ugmpgtbl_setentry(u32 gsid, u32 addr, u64 v) {
8 /* sysclog */ {v=v&7; v&=~_X; v|= _R; v|=_W;}
9 /* aprvexec */ {v=v&7; v&=~_W; v|= _R; v|=_X;}
10 /*@assert sysclog: (! (v&_X) && (v&_R) && (v&_W));*/
11 /*@assert aprvexec: (! (v&_W) && (v&_R) && (v&_X));*/
12 }

```

(b)

Fig. 3: Composition check: (a) **hyperdep** and **sysclog** üobjects both use ugmpgtbl üAPI setentry interface to set guest memory page protections in a composable manner. (b) **sysclog** and **aprveexec** both use setentry in a non-composable manner.

to soundly model interference from devices and other cores. Second, we preclude use of C function pointers and CASM indirect jump instructions, which remain challenging for current state-of-the-art static analyzers [21]. In practice (§7.2), this weakening does not stop us from verifying important security properties, since such properties are implemented via sequential üobjects using non-DMA memory.

7.2. üXMHF Verification – Verification of üXMHF consists of: (a) üobject composition check, and (b) verifying üSpark invariants (§5) and üobject local properties. Throughout this section we use *vh*, *uh* and *ug* as acronyms for verified and unverified hypervisor and unverified guest üobjects respectively.

7.2.1. üObject Composition Check: Resources accessed by multiple üobjects are guarded by üAPI üobjects (§4.3.1). Here we check that all üobjects are composable over the set of üAPIs they use. At a high level, this is checked by constructing an assertion that captures the conjunction of the possible values that the two üobjects write to a shared resource, and then verifying that this assertion is not violated. More specifically, for every üAPI üobject, an interface stub function is first created using its manifest. Next, the stub is populated with invariant definitions and assertions (if any) listed in the manifest of every *vh* and *uh* non-üAPI üobject that invokes it. Figure 3a shows an example stub for ugmpgtbl üobject setentry interface with **hyperdep** and **sysclog** hypapps enabled. Lines 2–6 are populated using the corresponding hypapp üobject manifests. Figure 3b shows the same stub with **sysclog** and **aprveexec** hypapps enabled. Finally, the assertions in the stub are verified under *non-deterministic* inputs. For example, **hyperdep** and **sysclog** both set the read, write and clear the execute bits for the memory protections of the provided guest memory-page (lines 2–3) and are therefore composable; the assertions (lines 4–6) in Figure 3a are valid. However, **sysclog** and **aprveexec** are not composable (Figure 3b) since **aprveexec** sets the execute bit while **sysclog** clears the execute bit in the protections for the provided memory-page (lines 9–

10). Note, such composition check assertions are also performed at runtime for üAPI invocations from *uh* üobjects (§4.3.1). This composition check procedure is üXMHF-specific, and a more general check is an interesting direction for future work.

7.2.2. *iObject Compositional Verification:* As we discussed in §5, we first verify üSpark invariants via a set of PAHs and specific POCs on all *vh* üobjects including the prime and sentinel. §7.2.3 describes POC verification in further detail. We then verify each of the üXMHF core, hypapp and üAPI üobjects for their local invariants. For brevity we summarize the **hyperdep** üobject verification approach here. Appendix B lists the invariants and verification approach for other üXMHF üobjects. **hyperdep** preserves the following invariant over the `ugmpgtbl` setentry üAPI: guest OS provided memory pages are marked read-write and not executable. We use deductive verification to verify the `hyperdep` üobject `activate` method to ensure that the guest page address that is passed is used as the parameter to the `ugmpgtbl` üobject `setentry` method with read, write and no-execute protections. Finally, we verify the üobject runtime library (§6.5) for memory safety including behavior specifications for the memory and string functions within `libuc`. Note, *uh* üobjects are not verified since their properties follow from üAPI invariants, ensured by our composition check (§7.2.1).

7.2.3. *POC Verification:* For brevity, we choose a sampling of POCs from a few üSpark invariants ($\text{Inv}_{\bar{u}}^4$, $\text{Inv}_{\bar{u}}^6$, $\text{Inv}_{\bar{u} \text{prog}}^6$, $\text{Inv}_{\bar{u} \text{prog}}^7$, and $\text{Inv}_{\bar{u}}^{10}$; see Appendix A and [73]) that showcase the importance of all the verification techniques described in §7.1.1. All the üSpark invariant POCs are verified using a combination of these techniques. Note that examples described below are necessary (but not sufficient since they are a sample) for the high-level proofs; for example the NESTEDCALL theorem (§5) cannot be proved if there is no non-overlapping, unity-mapped memory ($\text{Inv}_{\bar{u}}^4$) or DMA protection ($\text{Inv}_{\bar{u}}^6$).

Figure 4 shows a POC code snippet – from the *vh* üobject page-table setup function within prime – for $\text{Inv}_{\bar{u}}^4$ verified using deductive verification. ACSL *requires-assign-ensure* clause triples (lines 4–11) are used to specify function behavior. In this case they specify that every memory address in the page tables is disjoint with virtual-to-physical unity mapping. ACSL *loop invariant* clause allows specification of loops with data structure invariants (lines 17–25). Finally, ACSL *ghost variables* – C statements and variables only visible in specifications – are most notably used for modular reasoning of nested function calls. For example, line 28 invokes a support function for obtaining the memory protection of the specified memory address. This is aliased into a ghost variable which can then be used within the specification (line 29). In summary, the requires-

```

1 // @ ghost u64 gflags[SZ_PDPT*SZ_PDT*SZ_PT];
2 /*@
3 ...
4 requires \valid(vhpgtbl1t[0..(SZ_PDPT*SZ_PDT*SZ_PT)-1]);
5 ...
6 assigns vhpgtbl1t[0..(SZ_PDPT*SZ_PDT*SZ_PT)-1];
7 assigns gflags[0..(SZ_PDPT*SZ_PDT*SZ_PT)-1];
8 ...
9 ensures (\forallofull u32 x; 0 <= x < SZ_PDPT*SZ_PDT*SZ_PT ==>
10   ((u64)vhpgtbl1t[x] == (((u64)(x*SZB_4K)
11   & 0xFFFFFFFFFFFF000ULL) | (u64)(gflags[x]))));
12 */
13 void gp_setup_vhmempgtbl(void){
14   u32 i, spatype, slabid=XMHF_SLAB_PRIME;
15   u64 flags;
16 ...
17 /*@
18   loop invariant 0 <= i <= (SZ_PDPT*SZ_PDT*SZ_PT);
19   loop assigns gflags[0..(SZ_PDPT*SZ_PDT*SZ_PT)],spatype,
20   flags,i,vhpgtbl1t[0..(SZ_PDPT*SZ_PDT*SZ_PT)];
21   loop invariant \forallofull integer x; 0 <= x < i ==>
22   ((u64)vhpgtbl1t[x]) == (((u64)(x*SZB_4K)
23   & 0xFFFFFFFFFFFF000ULL) | (u64)(gflags[x]));
24   loop variant (SZ_PDPT*SZ_PDT*SZ_PT) - i;
25 */
26   for(i=0; i < (SZ_PDPT*SZ_PDT*SZ_PT); ++i){
27     spatype=_gp_getspatype(slabid, (u32)(i*SZB_4K));
28     flags=_gp_getptflags(slabid, (u32)(i*SZB_4K),spatype);
29     // @ ghost gflags[i] = flags;
30     vhpgtbl1t[i] = pae_make_pte( (i*SZB_4K),flags);
31   }
32 }
```

Fig. 4: Frama-C ACSL behavior specification and deductive verification: *vh* üobject memory page-table setup top-level function in prime.

```

prime.cs:
1 ...
2 ci_movel_eax_medi();
3 ...
hwm-cpu.c:
4 void ci_movel_eax_medi(){
5 ...
6 if(uhm_cpu_r_edi >= IMMUL0 && uhm_cpu_r_edi >= IMMUIH)
7   uhm_immuwr(uhm_cpu_r_edi, uhm_cpu_r_eax);
8 ...
9 }
hwm-iommu.c:
10 void _gxmhfhwm_iommu_wr(u32 addr, u32 val){
11 ...
12 if (addr==IMMUCTRL){ cbuhm_immuctrlwr(val); ... }
13 ...
14 }
prime-vdrv.c:
15 void cbuhm_immuctrlwr(u32 val){
16 //@assert !(val & IMMUTE) || (val & IMMUTE) &&
17 // gxmhfhwm_iommu_retaddr == (u32)&gp_ret;
18 }
19 ...
```

Fig. 5: üSpark hardware model and proving IOMMU DMA protection.

assigns-ensures clause triplet is sufficient to represent the function behavior, and the loop invariants and ghost variables within the function are used to prove the clause triplet. ACSL is highly expressive with global and type invariants, including first-order, polymorphic, recursive and higher-order specifications [25].

Fig 5 shows a POC code snippet for $\text{Inv}_{\bar{u}}^6$ verified using abstract interpretation and the hardware model. The snippet is part of the DMA protection setup function within prime. Line 2 in Fig 5 shows üobject using a designated CASM instruction to perform device I/O to the IOMMU. The hardware model hooks this CASM instruction to the IOMMU device model if the specified I/O range falls within the IOMMU device space (lines 6–7). The IOMMU modeling then simulates the required logic based on the register accessed and value written (line 12). The hardware model also invokes the appro-

Component	Impl (SLoC)	Annot	Verification		übp	ücasm	ücc	ümf	ühwm	ücvf	Total
			Time[s]	Mem[GB]	108	296	138	132	199	148	1021
<i>üObject libraries:</i>											
libuc	151	223	101	0.80							
libucrypt	88	58	35	0.05							
libustub	120	97	5	0.03							
libuhw	1706	749	465	0.90							
prime	2043	3176	1386	1.10							
sentinel	672	501	423	0.75							
<i>üXMHF üAPI üObjects:</i>											
ugmpgtbl	128	91	174	0.65							
ugcpust	73	46	118	0.70							
uhcpust	26	23	99	0.50							
<i>üXMHF Core üObjects:</i>											
xcstrt	97	0	53	0.12							
xcihub	247	202	147	0.60							
xcehub	41	0	48	0.08							
<i>üXMHF Hypapp üObjects:</i>											
sysclog	255	213	174	0.75							
sysclognw	1193	273	413	0.85							
hyperdep	161	31	98	0.70							
aprexec	199	—	—	—							
Total/Avg.	7200	5544	3739	0.57							
üSpark üAPI composition check			18	0.23							
üSpark Hardware model SLoC = 2079											

Fig. 6: üXMHF üobject SLoC and verification time/memory.

priate verification driver callbacks whenever such device registers are written to (line 12). This ensures required device state invariants. For example, assertions in lines 16–17 of the IOMMU control register callback ensure that DMA page-table protections when enabled always point to the populated DMA page tables (which are populated by the prime in a separate function not shown). This ensures that devices can only perform DMA to üobject DMA memory region. Similar techniques are used to: (a) hook designated CASM instructions for üobject access to system memory including *ug* üobject memory regions; and (b) proving intra-üobject CFI in the presence of both C and CASM functions by ensuring that CASM functions respect the C ABI and preserve callee registers and stack frames (via corresponding hardware model callbacks, assertions, and ACSL annotations).

POCs for $\text{Inv}_{\text{iprog}}^6$ and $\text{Inv}_{\text{iprog}}^7$ are verified by analysing the abstract syntax trees (AST) to preclude statements involving function pointers in C functions and to ensure CASM functions always end with a CASM `ret` instruction respectively. The POC for Inv_i^{10} is verified via CFG analysis to enforce üSpark blueprint conformance. Similar AST-based techniques are employed to: (a) embed hardware model statements, (b) substitute Assembly mnemonics, and (c) ensure soundness of the hardware model by precluding C functions from touching hardware model functions and variables and vice-versa.

8. EVALUATION

8.1. System size and Verification TCB – üXMHF is implemented in 7001 SLoC *verified* privileged code split

Fig. 7: Frama-C üSpark specific plugins are written in OCaml and build atop existing Frama-C kernel and standard plugins.

into 11 üobjects with 5544 lines of ACSL annotations and 2079 lines of hardware model (Figure 6). We also implemented an unverified hypervisor extension (**aprexec**; 199 SLoC) to illustrate how unverified and verified hypervisor üobjects interact. Depending on the properties, üobject verification takes 48 seconds to 23 minutes, and up to 1.1 GB of memory. Cumulative verification time is just over an hour, comparing favorably to related verification efforts [34]. Compositional verification enables each üobject to be (re-)verified separately. The prime üobject takes the longest to verify, but typically does not change as often as other üobjects. Decomposing prime into multiple üobjects can further reduce its (re-)verification time significantly.

Our verification TCB comprises the ACSL annotations, the hardware model (§7.1.2), and Frama-C with associated plugins. Modularity of üobject programs helps keep annotations small and feasible for manual review. Various orthogonal techniques exist to validate our hardware model [50], [58] that we plan to explore as future work. Frama-C is an industrial-strength tool used in many critical systems today [41]; we did not encounter any soundness bugs in these tools (§9). Frama-C üSpark specific plugins (totaling 1021 SLoC of OCaml; Figure 7) are modular, simple, and built upon the existing Frama-C kernel and plugins making them amenable to manual audits. Overall, our TCB compares favorably with other prior approaches (Figure 8).

8.2. Developer Effort – üXMHF was developed and verified in a year by a single system developer who was new to Frama-C/ACSL. A fraction of the time was spent adding implementation support for multiple hypapps with a greater part spent on porting to the üSpark hypervisor architecture by creating required üobjects and adding verification related harnesses and annotations. Annotation-to-code ratio (ACR) ranges from 0.2:1 to 1.6:1 (Figure 6). For üobjects whose properties rely solely on üAPI’s the ACR is small (e.g., **hyperdep**). üObjects with properties requiring functional correctness (e.g., **sysclog** and **xcihub**) have relatively larger ACR. The prime and sentinel üobjects have the highest ACR since they discharge most of the üSpark invariants.

8.3. Performance Measurements – All performance benchmarks were carried out on a Dell Optiplex 9020 with an Intel Core-i5 4590 quad-core processor with 4GB of memory. All üobjects were compiled with full compiler optimizations turned on.

8.3.1. üSpark Microbenchmarks: The cost of a CASM NULL function call is only 12 clock cycles. Sentinel

System/TCB	Compiler	HW Model	Annot./Specs.	Verification Tools	Other
Verve	In TCB	NS	NS	Boogie, BoogieASM, TAL checker, Z3	Iso-gen, boot-loader
sel4	In TCB	NS	In TCB	Isabelle/HOL, HOL4, Myreen, Sonolar, Z3	boot-loader
Hyper-V/Vcc	In TCB	In TCB	In TCB	Vcc, Boogie, Z3	boot-loader
Ironclad	Out-of TCB	In TCB	In TCB	Boogie, BoogieASM, Dafnyspec, Symdiff, Z3	None
mCertIKOS	Out-of TCB	NS	In TCB	Coq	None
üSpark	Out-of TCB	In TCB	In TCB	Frama-C, üSpark plugins, Z3, CVC3, Alt-Ergo	None

Fig. 8: Development and Verification Tools Trusted Computing Base (TCB) Comparison: All systems in addition employ a preprocessor (either built-in or stand-alone) for macro substitution and file inclusion and an assembler and linker to produce machine code; *NS* = *Not supported*

Verified- Verified	Verified-Unverified / Unverified-Verified				HVM
	SEG	CR3	TSK	HVM	
2x	37x	48x	70x	278x	

Fig. 9: üSpark Microbenchmarks: Sentinel üobject call overheads w.r.t regular NULL function call in privileged mode.

CPUID	RDMSR	WRMSR	XSETBV	CRx	VMCALL	SIP1
100	98	98	100	100	99	99

Fig. 10: üXMHF Microbenchmarks: core intercept handling clock-cycle latency as % of native XMHF performance without üSpark.

sysclog	hyperdep	aprveexec	ropdet	iousb	ionet	iodisk	ioser
97	99	91	89	95	96	99	99

Fig. 11: üXMHF Hypapp and I/O Benchmarks as % of native XMHF performance without üSpark.

SPEC	ioz-read	ioz-write	compbench	apache
100	100	100	100	100

Fig. 12: üXMHF Guest CPU and I/O Benchmarks as % of native XMHF Guest performance without üSpark.

call overhead for verified-to-verified üobject transitions is 2x w.r.t NULL function call (Figure 9). This is due to control transfers to the sentinel and üobject entry points and return addresses via JMP instructions. For transitions involving unverified üobjects the sentinel overhead is broken up into: (a) software overhead such as register saving, parameter marshalling, and call-policy enforcement; and (b) hardware deprivileging overhead. As seen, segmentation and CR3-based page tables provide the lowest overheads (37x and 48x), but are still an order of magnitude larger than the verified-to-verified sentinel call overhead. Hardware deprivileging adds a significant portion (upward of 60%) to the sentinel call in this case. These overheads are comparable to existing unverified disaggregated systems and micro kernels (§10).

8.3.2. üXMHF Microbenchmarks: For purposes of micro benchmarking we measure the üXMHF `xcihub` üobject, which handles several intercepts required for guest execution. üXMHF delivers near native XMHF performance in all cases (Figure 10). We attribute the small overhead for certain intercepts to the code refactoring using üobjects.

8.3.3. üXMHF Guest Benchmarks: We execute both compute-bound and I/O-bound applications for guest benchmarking purposes. For compute-bound applications, we use the SPEC-INT 2006 suite. For

I/O-bound applications, we use the iozone (disk reads and writes with 4K block size and 2GB file size), compilebench (project compilation benchmark), and Apache web server performance (ab tool with 200,000 transactions with 20 concurrent connections). üXMHF does not affect native XMHF’s guest performance in all cases (Figure 12).

8.3.4. üXMHF Application Benchmarks: We use the hypapps described in §6.1 along with another unverified hypapp `ropdet` (which captures guest branch information for ROP detection) for hypapp performance benchmarking. We wrote a guest üobject that interacts with the hypapps to leverage their services as follows. For `sysclog`, activate syscall logging by setting the syscall code page to no-execute and perform sample syscalls. For `hyperdep`, set a data page to no-execute and perform data read and write operations on that page. For `aprveexec`, setup a code page for approved execution, and invoke the hypapp to approve and lock the page against writes, before executing a sort function on that code page. Finally, for `ropdet`, register a test function over which ROP detection is to be performed, and invoke the test function to collect branch information. Figure 11 shows the performance overhead for these hypapps compared to native XMHF without üSpark. Verified `sysclog` and `hyperdep` run close to native XMHF speeds (2% avg. overhead). Unverified `aprveexec` and `ropdet` incur higher overheads (9% and 11% respectively). The overhead is due to üAPI invariant checks (<10%) and the sentinel cost of deprivileging, shadow stack and parameter marshalling (§8.3.1).

For I/O performance benchmarks, we wrote a mix of DMA I/O (usb and net) and programmed I/O operations (disk and serial) within a hypervisor üobject. The I/O performance overhead (Figure 11) is anywhere from 1-5% with the DMA-based I/O incurring more overhead. We attribute the higher DMA-based I/O overhead to the IOMMU page tables for DMA access. Note that üSpark does not actively interpose on any I/O operations, which results in a much lower overhead. These I/O overheads also match up to existing micro hypervisor I/O architecture overheads [67], [72], [86].

9. EXPERIENCE AND LESSONS LEARNED

9.1. Frama-C – The WP plugin’s limited casting support helped detect erroneous esoteric casts, e.g., pointer

to int/u8. While the Value plugin cannot propagate states to arbitrarily large loops, the semantic unrolling option helped propagate states only for desired functions so memory/time resources can be well spent. WP loop invariants are versatile in supporting unbounded loops with nesting. WP discharges proofs more effectively when operating over single-dimensional array accesses for mutating assignments and invariants and simple statements using shift and bit-wise operators. WP also caused proof failures in certain cases with local variable aliasing of function parameters; using parameter variables directly ameliorated the issue. We did not encounter any soundness bugs in Frama-C and its plugins.

9.2. Verification Theories – Automated verification results vary by theory, e.g., Alt-Ergo and Z3 failed to discharge a few verification conditions (VC) that CVC3 handled. Frama-C’s ability to combine provers was very useful; CVC3, Z3 and Alt-Ergo together solved all the VCs generated during verification.

9.3. Annotations – ACSL is versatile in its support for writing partial specifications (e.g., memory safety of SHA-1) and assertions as well as complete specifications (e.g., page-table setup). Further, ACSL annotations use actual C variables and operations. This expressivity spectrum thus allows system programmers to easily transition into the verification domain by initially using simple assertions and function contracts (partial specifications) and iteratively mastering complete specifications.

9.4. CompCert – The C99 subset handled by CompCert suffices to implement most systems-level software constructs. However, struct bit fields with packing and alignment within struct fields are currently unsupported. We added methods with bitwise operators to pack, unpack, deconstruct, and align such variables in the sources.

10. RELATED WORK

10.1. Unverified monolithic – SELinux [66], AppArmor [1] and FBAC [59] are some examples of OS kernel modifications that add features to an existing (privileged) kernel to enforce various access control policies. Such approaches suffer from the lack of assurance and separation: a bug in an extension or the core can exist, and then affect other parts of the system arbitrarily.

10.2. Unverified disaggregation – Xen/Xoar [17] converts Xen into depriveleged partitions. NOVA [67] depriveleges everything (including VMM modules), except for a small privileged micro kernel. Safe composition of OS kernel extensions include extensible operating systems [10], [15], [20], [23], [39], [61], kernel driver isolation [13], [28], [47], [48], [69], [70], [78], interposition mechanisms [29], [35], [37], [40] and API compatibility libraries [5], [7], [9], [30], [56], [79]. Xax [19] confines untrusted application code to an ABI for accessing OS services. SGX [4] protects application code from (buggy)

privileged code. Disaggregation brings mere isolation but no formal guarantees on its own.

10.3. Verified sandboxing – SFI [52], [54], [60], [76], [82] is a software-based approach for application-level memory isolation but lacks support for low-level privileged instructions and hardware device access, which are necessary for hypervisor and its extensions. Also, SFI employs unverified binary rewriting which can change the semantics of the program and break invariants necessary for compositional verification. Singularity [36] sacrifices legacy compatibility with a complete redesign of a OS written in type-safe languages (MSIL/TAL) and uses software mechanisms to isolate processes (SIP) and supports only memory and type-safety properties.

10.4. Verified kernels – seL4 [43] verifies full functional correctness of the C implementation (7500 LOC) of the micro kernel by showing that it refines an abstract specification. Their specifications don’t support abstractions among the kernel or the different kernel modules. These interdependencies often lead to more complex invariants which are difficult to prove (20 person years). Further, seL4 does not allow adding properties using untrusted services; such additions require direct integration into the kernel and lengthy re-verification. Furthermore, there is no support for Assembly (ASM) or device states, which precludes verification of low-level code interacting with devices; (1200 C and 500 ASM SLoC remain unverified). mCertiKOS [31] follows a similar approach to seL4 but makes the abstract specification layered to reduce the interdependencies among the kernel and various extensions and makes the verification process more tractable for an admittedly stripped down version of the original CertiKOS kernel (single-core, non-preemptible custom guest OS, basic process and syscall handling). There is no hardware model and support for ASM is limited to only general-purpose registers. Adding extra system instruction support and device models does not seem trivial; even the stripped down version of the kernel has 300 C and 170 ASM SLoC unverified. This is attributed to memory model limitations of their methodology [31]. Lastly, both mCertiKOS and seL4 require the developer to write line-for-line specifications for C/ASM code in a different abstract language (Isabelle/HOL or Coq/Ocaml/Lasm) with a very steep learning curve.

The VCC project [16], [45] verifies the functional correctness of a fixed Hyper-V hypervisor codebase running a multi-CPU guest, via automated theorem proving. However, the code annotations do not support abstractions among the core hypervisor or drivers. This leads to complex invariants due to interdependencies; only 20% of the hypervisor code-base has been verified [16]. Further, their ASM verification methodology and lack of a full hardware model only allows proving memory safety and arithmetic properties for ASM functions while

precluding compiler optimizations for the corresponding C callee functions [51]. XMHF [72] employs the CBMC model checker with assertions on the C code of a micro-hypervisor to verify memory integrity. However, multiple extensions or composing other properties on top of memory integrity are unsupported. Further, that effort *assumes* interface confinement and leaves out 422 C and 388 ASM SLoC due to limitations of CBMC with large-loops and lack of a hardware model.

10.5. Verified System Stack – In Verve [81], a simplified OS and applications are verified for type and memory safety using a Hoare-style verification condition (VC) generator and automated theorem proving. Ironclad [33] extends Verve with support for higher-level application properties. High-level specifications (written in Dafny) are translated to corresponding code with VCs discharged via an automated theorem prover; the verification took 3 person-years. Verisoft [6] integrates hardware and software, with high-level specifications written in C0 (a tiny subset of C semantics) and refined down to a custom CPU semantics. The verification took 20 person-years on a simple OS with only a disk driver. System stack verification approaches, while powerful, sacrifice compositionality, legacy compatibility and performance. Any changes to kernel code and/or extension configuration requires lengthy re-verification (in person years). Further, the entire system software stack has to be re-implemented in type-safe languages such as C# and TAL (in Verve) or in high-level Dafny specifications (in Ironclad) or on a non-commodity CPU abstraction (in Verisoft). Furthermore, these approaches lack support for co-existence with unverified programs or a guest OS.

11. LIMITATIONS AND FUTURE WORK

We now discuss current limitations of our approach with pointers to future work towards bridging these gaps.

11.1. Hardware Model – Our hardware model is currently a trusted component. However, orthogonal techniques such as path-exploration lifting [50] and mechanized x86-multiprocessor semantics [58] provide a solid foundation on which we plan to build upon and validate our hardware model in the future.

11.2. CASM and Certified Compilation – Our high-level proofs depend on CompCert’s specification of the C memory and register semantics and CASM’s adherence to those semantics (discharged as invariants on the source-code and our hardware model) to ensure that the C and Assembly code operate on disjoint state. In the future, we plan on leveraging recent developments with CompCert such as the ability to compile and link multi-module source programs [68] to cleanly extend the bi-simulation proof of the CompCert compiler to encompass hardware state and Assembly code. Future work also involves proving (e.g., via bi-simulation) the

semantic equivalence between the hardware model and the corresponding Assembly instructions and demonstrating the semantic synergy between CompCert, CASM and the Frama-C kernel more rigorously for proved properties to translate to the binary.

11.3. Functional Verification – Our focus in this paper is on security invariants and trace properties and functional correctness to support such properties. We are optimistic that liveness properties and full-functional correctness are achievable future goals and not any more harder than existing approaches [31], [33], [43].

11.4. Concurrency – We have shown that a practical multi-threaded system with interesting security properties can be built by dealing with a serialized execution model and sequential verification in lieu of complex concurrent verification. However, we do realize the importance of relaxing our serialized execution model especially in high-performance computing environments and plan on leveraging source-level multi-threaded verification (e.g., Frama-C mthread plugin [24]) to address concurrency in the future.

11.5. Soundness of Tools – Similar to existing approaches, we assume that the verification tools such as Frama-C with associated plugins and back-end theorem provers such as Z3, CVC3 and Alt-Ergo are sound (§8.1,§3.3). Discharging this assumption, while a desirable goal, is currently an open and hard problem in the face of formal methods. However, seminal breakthroughs such as certified software model-checking [55] and formal verification of C static analyzers [38] give us hope that proving soundness of our verification tools will indeed be possible in the future.

11.6. Applicability – Our future work involves generalizing überSpark to a more broadly applicable framework for building compositionally verifiable systems. We are exploring the applicability of überSpark to general-purpose hypervisors (e.g., Xen and KVM), BIOS, device firmware, operating-system kernel and drivers, user-space applications and browser extensions including vertical integration among these stacked subsystems. The immediate challenges we envision there include unraveling complex data structures, supporting dynamic memory allocations and use of indirect function calls in addition to supporting some form of concurrency.

12. CONCLUSION

We presented überSpark, an innovative architecture enforcing verifiable object abstractions in low-level C and Assembly languages and leveraging them in combination with off-the-shelf C software verifiers and certifying compilers to produce high assurance hypervisors for commodity platforms. We incrementally developed and verified a commodity x86 micro-hypervisor using überSpark, and performed a comprehensive evaluation which

shows automated compositional verification with modest development effort and minimal runtime overhead.

Availability: ÜBERSPARK and ÜXMHF sources are available at: <http://uberspark.org>

Acknowledgements: We thank the anonymous reviewers for their detailed comments. We also thank Úlfar Erlingsson, Martín Abadi and Matt Loring for their feedback and insights. This work was partially supported by the Intel Science and Technology Center for Secure Computing, AFOSR MURI on Science of Cybersecurity, the NSA/CMU Science of Security Lablet, and the NSF CNS-1018061 grant. Copyright 2016 CyLab and CMU².

REFERENCES

- [1] Novell, AppArmor, and SELinux Comparison. http://www.novell.com/linux/security/apparmor/selinux_comparison.html.
- [2] CVE-2008-3687: Heap-based buffer overflow in Xen 3.3, when compiled with the XSM:FLASK module, allows unprivileged domain users (domU) to execute arbitrary code via the flaskop hypercall. <https://cve.mitre.org/>, 2008.
- [3] VMSA-2009-0006: VMware patches for ESX and ESXi resolve a critical security vulnerability. <http://www.vmware.com/security/advisories/>, 2009.
- [4] Software Guard Extensions Programming Reference 329298-001. <http://software.intel.com>, 2013.
- [5] <http://cygwin.com>, 2014.
- [6] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. *J. Autom. Reasoning*, 42(2-4):389–454, 2009.
- [7] J. Appavoo, M. Auslander, D. Edelsohn, D. D. Silva, O. Krieger, M. Ostrowski, B. Rosenberg, R. W. Wisniewski, and J. Xenidis. Providing a linux api on the scalable k42 kernel. In *ATC*, 2003.
- [8] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004. Workshop on Formal Techniques for Java-like Programs (FTfJP), ECOOP 2003.
- [9] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing os extensions safely and efficiently with bascule. In *Proc. of EuroSys*, 2013.
- [10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proc. of SOSP*, 1995.
- [11] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM*, 2006.
- [12] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *In Proc. of IEEE ARITH*, 2013.
- [13] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *TEC*, 2010.
- [14] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar. Towards verifiable resource accounting for outsourced computation. In *ACM VEE*, 2013.
- [15] D. R. Cheriton and K. J. Duda. A caching model of os kernel functionality. In *OSDI*, 1994.
- [16] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOLS*, 2009.
- [17] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proc. of SOSP*, 2011.
- [18] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. of CCS*, 2008.
- [19] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proc. of OSDI*, 2008.
- [20] D. R. Engler, M. F. Kaashoek, and J. O’Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of SOSP*, 1995.
- [21] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proc. of CCS*, 2015.
- [22] A. Fattori, R. Paleari, L. Martignoni, and M. Monga. Dynamic and transparent analysis of commodity production systems. In *Proc. of IEEE/ACM ASE 2010*, 2010.
- [23] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of OSDI*, 1996.
- [24] Frama-C. Mthread plug-in. <http://frama-c.com/mthread.html>, 2012.
- [25] Frama-C Team. ACSL: ANSI/ISO C Specification Language v1.9. <http://www.frama-c.com>, 2015.
- [26] J. Franklin, S. Chaki, A. Datta, and A. Seshadri. Scalable Parametric Verification of Secure Systems: How to Verify Reference Monitors without Worry-

²This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003658

- ing about Data Structure Size. In *IEEE S&P*, 2010.
- [27] J. Franklin, A. Seshadri, N. Qu, S. Chaki, and A. Datta. Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor. Technical Report CMU-CyLab-08-008, CMU CyLab, 2008.
- [28] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *ASPLOS*, 2008.
- [29] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. Slic: An extensibility system for commodity operating systems. In *ATC*, 1998.
- [30] D. Given. <http://lbw.sf.net/>, 2010.
- [31] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proc. of POPL*, 2015.
- [32] N. Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, Oct. 1985.
- [33] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proc. of OSDI*, 2014.
- [34] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proc. of OSDI*, 2014.
- [35] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *WINSYM*, 1999.
- [36] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.
- [37] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. *SIGOPS Oper. Syst. Rev.*, 27(5):80–93, Dec. 1993.
- [38] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified c static analyzer. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, pages 247–259, New York, NY, USA, 2015. ACM.
- [39] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. of SOSP*, 1997.
- [40] Y. A. Khalidi and M. N. Nelson. Extensible file systems in spring. *SIGOPS Oper. Syst. Rev.*, 27(5):1–14, Dec. 1993.
- [41] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *FAC*, 27(3):573–609, 2015.
- [42] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, Feb. 2014.
- [43] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. of SOSP*, 2009.
- [44] K. Kortchinsky. Cloudburst: A VMware guest to host escape story. Black Hat, 2009.
- [45] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM*, 2009.
- [46] X. Leroy. Formal certification of a compiler backend, or: programming a compiler with a proof assistant. In *Proc. of POPL*, 2006.
- [47] B. Leslie, P. Chubb, N. Fitzroy-dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. In *Journal of Computer Science and Technology*, 2005.
- [48] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*, 2004.
- [49] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proc. of USENIX Security*, 2008.
- [50] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for Lo-fi emulators. *SIGPLAN Not.*, 47(4):337–348, Mar. 2012.
- [51] S. Maus, M. Moskal, and W. Schulte. Vx86: x86 assembler simulated in C powered by automated theorem proving. In *Proc. of AMAST*, 2008.
- [52] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security*, 2006.
- [53] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P*, May 2010.
- [54] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. Rocksalt: Better, faster, stronger SFI for the x86. *SIGPLAN Not.*, 47(6):395–404, 2012.
- [55] K. S. Namjoshi. Certifying model checkers. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, pages 2–13, 2001.
- [56] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Ollinsky, and G. C. Hunt. Rethinking the library os from the top down. *SIGARCH Comput. Archit. News*, 39(1):291–304, Mar. 2011.
- [57] D. Quist, L. Liebrock, and J. Neil. Improving antivirus accuracy with hypervisor assisted analysis. *J. Comput. Virol.*, 7(2), May 2011.
- [58] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-cc multiprocessor machine code. *SIGPLAN*, 44(1):379–391, 2009.
- [59] Z. C. Schreuders, C. Payne, and T. McGill. Techniques for automating policy specification for application-oriented access controls. In *Proc. of*

- ARES, 2011.
- [60] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proc. of USENIX Security*, 2010.
 - [61] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of OSDI*, 1996.
 - [62] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of SOSP*, 2007.
 - [63] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: A fast capability system. In *SOSP, SOSP '99*, pages 170–185, 1999.
 - [64] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proc. of CCS*, 2009.
 - [65] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *EuroSys*, 2006.
 - [66] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux LSM. *NSA*, 2001.
 - [67] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proc. of Eurosys*, 2010.
 - [68] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional compcert. In *POPL*, pages 275–287, 2015.
 - [69] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. of SOSP*, 2003.
 - [70] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM TOCS*, 23(1):77–110, Feb. 2005.
 - [71] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.
 - [72] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proc. of IEEE S&P*, 2013.
 - [73] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überSpark: Enforcing Verifiable Object Abstractions for Compositional Security Analysis of a Hypervisor. Technical Report CMU-CyLab-16-003, CMU CyLab, 2016.
 - [74] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In *Proc. of TRUST*, 2012.
 - [75] A. Vasudevan, N. Qu, and A. Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proc. of IEEE HICSS*, 2011.
 - [76] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SIGOPS OSR*, 27(5):203–216, Dec. 1993.
 - [77] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proc. of EuroSys 2012*, 2012.
 - [78] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, 2008.
 - [79] Wine. <http://www.winehq.org/>, 2014.
 - [80] X. Xiong, D. Tian, and P. Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. In *Proc. of NDSS*, 2011.
 - [81] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proc. of PLDI*, 2010.
 - [82] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. of IEEE S&P*, 2009.
 - [83] M. Yu, V. D. Gligor, and Z. Zhou. Trusted display on untrusted commodity platforms. In *ACM CCS*, pages 989–1003, 2015.
 - [84] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. of SOSP*, 2011.
 - [85] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE S&P*, 2012.
 - [86] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with Giants: Wimpy Kernels for On-demand Isolated I/O. In *Proc. of IEEE S&P*, 2014.

APPENDIX A ÜSPARK INVARIANTS

ÜSpark reasoning relies foundationally on a set of invariants – properties that must hold throughout the execution of a üSpark hypervisor. The invariants are divided into üSpark system invariants (Figure 13) and üSpark general programming invariants (those that pertain specifically to üSpark üobject C and CASM functions; Figure 14). Each invariant is proved by reducing it further to a set of *proof-assumptions on hardware* (PAHs) and *proof-obligations on code* (POCs) using the üSpark blueprint (üBP; §4–Figure 2). POCs are then discharged on all üSpark verified üobjects including the prime and sentinel using specific verification tools and techniques (§7). A hypervisor implementation is compliant with üSpark– and therefore amenable to compositional reasoning – if it satisfies all the üSpark invariants. Full details including a formal model of the üSpark architecture, semantics, verification approach, associated theorem proofs and invariant-to-PAH/POC mappings can be found in our technical report [73].

$\text{Inv}_\text{ü}^1$	üSpark begins execution with the entry point of a distinguished initial “prime” üobject s_1 in single-core mode with just core 1 activated
$\text{Inv}_\text{ü}^2$	A special “asynchronous” function $\text{startcores}(s)$ activates all cores $i > 1$ and begins executing a designated üobject s immediately thereafter; all cores remain active thereafter for the system lifetime.
$\text{Inv}_\text{ü}^3$	Asynchronous control transfers (hardware interrupts, exceptions and intercepts) respect the blueprint state execution threading and transitions
$\text{Inv}_\text{ü}^4$	üObject memory regions are unity-mapped and non-overlapping
$\text{Inv}_\text{ü}^5$	üObject s accesses only its own memory
$\text{Inv}_\text{ü}^6$	üObject code, data and stack regions are DMA protected
$\text{Inv}_\text{ü}^7$	üObject code is write-protected
$\text{Inv}_\text{ü}^8$	Inter-üobject synchronous control-flow respect blueprint transitions
$\text{Inv}_\text{ü}^9$	Each core has its own stack at all times and stays within the stack limits.
$\text{Inv}_\text{ü}^{10}$	Blueprint state has state appropriate execution threading (multi-core or single-core)
$\text{Inv}_\text{ü}^{11}$	Locks behave like “memory fences”; any write preceding a call to unlock is observed by any read following the next call to lock

Fig. 13: üSpark System Invariants

$\text{Inv}_{\text{üprog}}^1$	CASM functions preserve caller registers
$\text{Inv}_{\text{üprog}}^2$	CASM functions establish local stack frame non-overlapping with incoming caller stack frame
$\text{Inv}_{\text{üprog}}^3$	CASM functions have conditional and unconditional branches local to the function
$\text{Inv}_{\text{üprog}}^4$	CASM functions establish callee incoming stack frame for calls to other C or CASM functions
$\text{Inv}_{\text{üprog}}^5$	CASM functions tear down local stack frame before returning
$\text{Inv}_{\text{üprog}}^6$	CASM functions end with return instruction
$\text{Inv}_{\text{üprog}}^7$	No function pointers in C functions
$\text{Inv}_{\text{üprog}}^8$	C and CASM functions do not write to caller stack frame params and return-address
$\text{Inv}_{\text{üprog}}^9$	CASM functions can only encode instructions within the domain of CASM instruction set
$\text{Inv}_{\text{üprog}}^{10}$	CASM non-local control transfer instructions can only be to fixed function entry points

Fig. 14: üSpark Programming Invariants

APPENDIX B VERIFICATION OF ÜXMHF ÜOBJECTS LOCAL INVARIANT PROPERTIES

We now describe our verification approach in detail for verifying the invariant properties of the üXMHF üobjects shown in Figure 15. For all the üobjects we verify via deductive verification that the üobject entry point function transfers control to the appropriate method handler for a given public method.

We verify the üAPI üobjects via abstract interpretation. For the `uhcpust` üobject we verify that the `write` method, in case of a write to MSR EFER, always preserves the EFER bits required for üSpark functionality.

üObject	Type	Invariant Property
<code>xcihub</code>	<i>vh</i>	On intercept invoke corresponding hypapp handler
<code>ugcpust</code>	<i>vh</i>	Writes to host state only by prime or sentinel
<code>uhcpust</code>	<i>vh</i>	No writes to host MSR EFER
<code>ugmpgtbl</code>	<i>vh</i>	No mapping of hypervisor memory regions
<code>hyperdep</code>	<i>vh</i>	Guest OS provided memory-pages are marked read-write and not executable
<code>sysclog</code>	<i>vh</i>	On system call trap intercept, log syscall information to network log buffer
<code>sysclognw</code>	<i>vh</i>	Log info in network log buffer and transmit buffer when full
<code>aprvexec</code>	<i>uh</i>	Guest OS approved code pages are always marked read-only and executable

Fig. 15: üXMHF Core, üAPI and Hypapp üobject invariants; *vh* = verified hypervisor üobject, *uh* = unverified hypervisor üobject

For the `ugmpgtbl` üobject we verify that the `setentry` method’s entry parameter does not fall within hypervisor memory regions. Finally, for the `ugcpust` üobject we verify that the `write` method disallows writes to any host-specific state in the guest VMCS.

For the `xcihub` üobject we employ deductive verification to verify the `main` method such that, for any given intercept a special function `hcbinvoke` is called with the intercept type and associated parameters. `hcbinvoke` is then verified to ensure that it calls all the registered hypapp üobjects for that intercept.

For the `sysclog` üobject we employ deductive verification to first verify that the `init` method invokes the `ugmpgtbl` üobject `setentry` method with the syscall page address with read and no-execute protections. We then verify that the syscall trap handler obtains syscall information via a call to the `ugcpust` üobject `read` method and stores this information to the network log buffer via a call to the `sysclognw` üobject `log` method.

We verify the `sysclognw` üobject via deductive verification and abstract interpretation. We use deductive verification to verify the `log` method to ensure that: (a) the buffer passed in as parameters is stored in the network buffer data structure, and (b) when the buffer is full, its contents are copied into the üobject `dmadata` region, buffer is reset, and the network send function is invoked. We then verify the `send` function via abstract interpretation to ensure that it programs the network card hardware to read from the `dmadata` region, transmit the buffer, and wait for end of transmission signal.

We use deductive verification to verify the `hyperdep` üobject `activate` method to ensure that the guest page address that is passed is used as the parameter to the `ugmpgtbl` üobject `setentry` method with read, write and no-execute protections.

Note, `aprvexec` (unverified) üobject is not verified since its properties follow from the `ugmpgtbl` üAPI invariants ensured by our composition check as described in §7.2.1.

Undermining Information Hiding (And What to do About it)

Enes Göktaş¹ Robert Gawlik² Benjamin Kollenda² Elias Athanasopoulos¹

Georgios Portokalidis³ Cristiano Giuffrida¹ Herbert Bos¹

¹ Computer Science Institute, Vrije Universiteit Amsterdam, The Netherlands

² Horst Görtz Institut for IT-Security (HGI), Ruhr-Universität Bochum, Germany

³ Department of Computer Science, Stevens Institute of Technology

Abstract

In the absence of hardware-supported segmentation, many state-of-the-art defenses resort to “hiding” sensitive information at a random location in a very large address space. This paper argues that information hiding is a weak isolation model and shows that attackers can find hidden information, such as CPI’s SafeStacks, in seconds—by means of *thread spraying*. Thread spraying is a novel attack technique which forces the victim program to allocate many hidden areas. As a result, the attacker has a much better chance to locate these areas and compromise the defense. We demonstrate the technique by means of attacks on Firefox, Chrome, and MySQL. In addition, we found that it is hard to remove all sensitive information (such as pointers to the hidden region) from a program and show how residual sensitive information allows attackers to bypass defenses completely.

We also show how we can harden information hiding techniques by means of an Authenticating Page Mapper (APM) which builds on a user-level page-fault handler to authenticate arbitrary memory reads/writes in the virtual address space. APM bootstraps protected applications with a minimum-sized safe area. Every time the program accesses this area, APM authenticates the access operation, and, if legitimate, expands the area on demand. We demonstrate that APM hardens information hiding significantly while increasing the overhead, on average, 0.3% on baseline SPEC CPU 2006, 0.0% on SPEC with SafeStack and 1.4% on SPEC with CPI.

1 Introduction

Despite years of study, memory corruption vulnerabilities still lead to control-flow hijacking attacks today. Modern attacks employ code-reuse techniques [9, 34] to overcome broadly deployed defenses, like data-execution prevention (DEP) [5] and address-space layout randomization (ASLR) [30]. Such attacks are still possible primarily because of address leaks, which are used

to discover the location of useful instruction sequences, called gadgets, that can be chained together to perform arbitrary computations [34].

In response, researchers have been exploring various directions to put an end to such attacks. A promising solution is code-pointer integrity (CPI) [24] that aims to prevent the hijacking of code pointers, and therefore taking control of the program. The separation of code pointers from everything else can be done by employing hardware or software-enforced isolation [39, 42], or by *hiding* the region where pointers are stored, which is a faster alternative, than software-based isolation, when hardware-based isolation is not available. This *information hiding (IH)* is achieved by placing the area where code pointers are stored at a random offset in memory and ensuring that the pointer to that area cannot be leaked (e.g., by storing it in a register). For example, safe versions of the stack, referred to as safe stacks, that only include return addresses are protected this way both by CPI and ASLR-guard [26]. This type of IH is also adopted by other defenses [7, 15] that aim to prevent attacks by eliminating data leaks, which would enable the location of gadgets, while it has also been adopted in shadow stack [13, 41] and CFI [27, 43] research.

Reliance on information hiding is, however, problematic. Recently published work [18] developed a memory scanning technique for client applications that can survive crashes. It exploits the fact that browsers, including Internet Explorer 11 and Mozilla Firefox, tolerate faults that are otherwise critical, hence, enabling memory scanning to locate “hidden” memory areas. Before that researchers demonstrated that it was possible to locate CPI’s safe region, where pointers are stored [17], if IH is used instead of isolation.

In this paper, we reveal two new ways for defeating information hiding, which can be used to expose the “hidden” critical areas used by various defenses and subvert them. The first is technique caters to multithreaded applications, which an attacker can cause a process to spawn

multiple threads. Such applications include browsers that now support threads in Javascript and server applications that use them to handle client connections. By causing an application to spawn multiple threads, the attacker “sprays” memory with an equal number of stacks and safe stacks. As the address space fills with these stacks, the probability of “striking gold” when scanning memory increases dramatically. We incorporate this technique, which we coin *thread spraying*, in the memory scanning attack described above [18] and show that we can locate safe regions, such as the safe stacks used by CPI and ASLR-guard and parallel shadow stacks [14], in *seconds* instead of tens of minutes. The second approach utilizes bookkeeping data of various standard libraries in Linux such as the POSIX threads library and glibc. Our investigation reveals several pointers that can lead to safe regions in information kept to manage thread local storage (TLS) and thread-control blocks (TCB). Isolating these leftover pointers with a better implementation might be possible. However, at the time of writing, there is no algorithm for assessing if *all sensitive* pointers are properly removed. We therefore argue that a sound implementation which excludes all pointers that can lead to a safe region from the rest of the process is challenging.

These two new attack vectors demonstrate that it is surprisingly hard to use information hiding to replace strong isolation. Nevertheless, it is possible to further harden IH and mitigate these attacks. We propose and develop a technique that aims to shift the odds back in favor of the defender. Our solution is based on two observations: first, only a small part of the memory allocated to safe areas is actually used, and, second, safe areas are usually accessed using well-defined instrumentation-induced instruction patterns that are distinguishable from application code. For example, under CPI only a small part of the safe region and stacks allocated by CPU is actually used, and, on x86-64 architectures, the `gs` segment register and the `RSP` register are used to access the safe region and safe stacks, respectively.

Building on these observations, we introduce APM (Authenticating Page Mapper), an IH hardening strategy that verifies that all the first accesses to inactive pages in safe areas follow legitimate and unforgeable execution patterns, increasing in this fashion the entropy of IH. APM is implemented as a user-level page-fault handler, exploiting the same page “trapping” technique used by mechanisms like checkpoint/restart [31], hot patching [4], and ATL+DEP [1]. To further improve security, APM artificially inflates the virtual memory region reserved for safe areas. In particular, it decreases the number of accessible pages on a shadow stack to the minimum, but reserves many (initially inaccessible) additional pages that serve as tripwire. All illegitimate accesses to non-accessible pages trigger an alarm. Like-

wise, we offset the stack pointer by a random number of bytes.

In summary, we make the following contributions:

1. We introduce a new attack vector, called thread spraying, that reduces the entropy of information hiding techniques by creating many threads and filling up the memory space with safe areas such as safe stacks. We demonstrate the practicality of thread spraying for browsers (Firefox and Chrome) and the MySQL server, all protected with CPI’s SafeStack, and show that we can find the hidden stacks in seconds.
2. We analyze in detail the strength of information hiding techniques used in state-of-the-art defenses and show that existing solutions are not free of pointers to safe areas. Our results enable an adversary to utilize these pointers to compromise the safe area.
3. We propose a new IH hardening strategy to improve information hiding by means of a user-level page fault handler that increases the entropy significantly, makes pages accessible on demand, and vets all first-time accesses to pages—greatly increasing the probability of detecting an attack. We evaluate the solution using a variety of applications as well as the SPEC benchmarks and show that the performance impact is very low (on average 0.3% on baseline SPEC, 0.0% on SPEC with SafeStack, 1.4% on SPEC with full CPI and barely measurable in browser benchmarks).

2 Threat Model

In this paper, we assume a determined attacker that aims at exploiting a software vulnerability in a program that is protected with state-of-the-art defenses (e.g., CPI [24] or ASLR-Guard [26]), and that has the capabilities for launching state-of-the-art code-reuse attacks [11, 16, 20]. We also assume that the attacker has a strong primitive, such as an arbitrary read and write, but the arbitrary read should *not* be able to reveal the location of a code pointer that could be overwritten and give control to the attacker, *unless* the safe area is somehow discovered. Under this threat model, we discuss in Sections 3 and 4 a number of possible strategies that can leak the safe area to the attacker. Later in this paper, we propose to harden IH using a technique that can effectively protect the safe area with a small and practical overhead.

3 Background and Related Work

In the following, we review relevant work on information hiding. We discuss both attacks and defenses to provide an overview of related work and hint at potential weaknesses. We show that prior work has already bypassed

several IH approaches, but these attacks all targeted defenses that hide very large areas (such as the 2^{42} byte safe area in CPI [17], or all kernel memory [22]). It is a common belief that smaller regions such as shadow stacks are not vulnerable to such attacks [26]. Later, we show that this belief is not always true.

3.1 Information Hiding

Many new defenses thwart advanced attacks by separating code pointers from everything else in memory. Although the specifics of the defenses vary, they all share a common principle: they must prevent malicious inputs from influencing the code pointers (e.g., return addresses, function pointers, and VTable pointers). For this reason, they isolate these pointers from the protected program in a *safe area* that only legitimate code can access in a strictly controlled fashion. In principle, software-based fault isolation (SFI [39]) is ideal for applying this separation. However, without hardware support, SFI still incurs nontrivial performance overhead and many defenses therefore opted for (IH) as an alternative to SFI. The assumption is that the virtual address space is large enough to *hide* the safe area by placing it in a random memory location. Since there is no pointer from the protected program referencing explicitly the safe area, even powerful information disclosure bugs [35] are useless. In other words, an attacker could potentially leak the entire layout of the protected process *but not* the safe area.

In recent years, this topic received a lot of attention and many systems emerged that rely (at least optionally) on IH. For example, Opaque CFI [27] uses IH for protecting the so called *Bounds Lookup Table* (BLT) and Oxymoron [7] uses IH for protecting the *Randomization-agnostic Translation Table* (RaTTle). Isomeron [15] needs to keep the *execution diversifier data* secret while StackArmor [41] isolates particular (potentially vulnerable) stack frames. Finally, CFCI [44] needs to hide, when segmentation is not available, a few MBs of protected memory. Although all these systems rely on IH for a different purpose, they are vulnerable to memory scanning attacks which try to locate these regions in a brute-force manner (as shown in Section 4). Since the Authenticating Page Mapper that we propose in this paper hardens IH in general, it directly improves the security of all these systems—irrespective of their actual goal.

3.2 ASLR and Information Leaks

Arguably the best known IH technique is regular *Address Space Layout Randomization* (ASLR). Coarse-grained ASLR is on by default on all major operating systems. It randomizes memory on a per-module basis. Fine-grained ASLR techniques that additionally randomize

memory on the function and/or instruction level were proposed in the literature [19, 29, 40], but have not received widespread adoption yet.

In practice, bypassing standard (i.e., coarse-grained, user-level) ASLR implementations is now common. From an attacker’s point of view, disclosing a single pointer that points into a program’s shared library is enough to de-randomize the address space [36]. Even fine-grained ASLR implementations cannot withstand sophisticated attacks where the attacker can read code with memory disclosures and assemble a payload on the fly in a JIT-ROP fashion [35].

For kernel-level ASLR, we view kernel memory as another instance of information to hide. From user space, the memory layout of the kernel is not readable and kernel-level ASLR prevents an attacker from knowing the kernel’s memory locations. However, previous work showed that it is possible to leak this information via a timing side channel [22].

In general, leaking information by abusing side channels is a viable attack strategy. Typically, an attacker uses a memory corruption to put a program in such a state that she can infer memory contents via timings [10, 17, 33] or other side channels [8]. This way, she can even locate safe areas to which no references exist in unsafe memory.

In the absence of memory disclosures, attackers may still bypass ASLR using Blind ROP (BROP) [8], which can be applied remotely to servers that fork processes several times. In BROP, an attacker sends data that causes a control transfer to another address and then observes how the service reacts. Depending on the data sent the server may crash, hang, or continue to run as normal. By distinguishing all different outcomes, the attacker can infer what code executed and identify ROP gadgets.

In this paper, we specifically focus on safe stacks (which are now integrated in production compilers) and review recent related solutions below.

3.3 Code-Pointer Integrity (CPI)

CPI is a safety property that protects all direct and indirect pointers to code [24]. CPI splits the address space in two. The normal part and a significantly large safe area that stores all code pointers of the program. Access to the safe area from the normal one is only possible through CPI instructions. Additionally, CPI provides every thread with a shadow stack, namely SafeStack, beyond the regular stack. The SafeStack is used for storing return addresses and proven-safe objects, while the regular stack contains all other data. SafeStacks are relatively small but they are all contained in a large safe area, which is hidden at a random location in the virtual address space.

Evans et al. showed how to circumvent CPI and find the safe area by probing using a side channel [17]. Depending on how the safe area is constructed, this attack may require the respawn-after-a-crash property to pull off the attack. This property is only available in (some) servers. Moreover, it is fragile, as it is very easy for an administrator to raise an alarm if the server crashes often. In Section 4, we will introduce an attack that demonstrates how we can efficiently locate CPI’s SafeStack in the context of web browsers.

3.4 ASLR-Guard

ASLR-Guard [26] is a recent defense that aims at preventing code-reuse attacks by protecting code addresses from disclosure attacks. It does so by introducing a secure storage scheme for code pointers and by decoupling the code and data regions of executable modules. A core feature is its shadow stack that it uses to separate completely the code pointers from the rest of the data. To efficiently implement this idea, again two separate stacks are used. First, the so called AG-stack which holds only code addresses is used by function calls and returns, exception handlers, etc. The second stack is used for any data operation and ensures that all code pointers and pointers to the AG-stack are encrypted. As a result, an adversary has no way of leaking the location of code images. We discuss the security of this design in Section 4.4.

3.5 Discussion

Information hiding has grown into an important building block for a myriad of defenses. While several attacks on the randomization at the heart of IH are described in the literature, it is still believed to be a formidable obstacle, witness the growing list of defenses that rely on it. Also, since the attacks to date only managed to find secret information occupying a large number of pages, it seems reasonable to conclude, as the authors of ASLR-Guard [26] do, that smaller safe areas are not so vulnerable to probing attacks. In this paper, we show that this is not always true.

4 Breaking Modern Information Hiding

In this section, we introduce two approaches towards uncovering the hidden information. First, we show however careful developers of IH approaches are, pointers to the safe area may still be unexpectedly present in the unsafe area. While this may not represent fundamental problems, there are other issues. Specifically, we show that an attacker may significantly reduce the large randomization entropy for secret data like shadow stacks by making the program spawn many threads in a controlled way, or corrupting the size of the stacks that the program spawns.

4.1 Neglected Pointers to Safe Areas

Safe stack implementations are an interesting target for an attacker and the ability to locate them in a large virtual address space would yield a powerful attack primitive. As an example, consider CPI’s SafeStack implementation that is now available in the LLVM compiler toolchain. Recall that the safe stack implementation of CPI moves any potential unsafe variables away from the native stack to make it difficult to corrupt or to gather the exact address of that stack. Any references to the safe stack in global memory that the attacker could leak would therefore break the isolation of SafeStack. Ideally for an attacker, such pointers would be available in program-specific data structures, but we exclude this possibility here and assume that no obvious information disclosure attacks are viable. However, even though the authors diligently try to remove all such pointers, the question is whether there are any references left (e.g., in unexpected places).

For this reason, we analyzed the implementation and searched for data structures that seemed plausible candidates for holding information about the location of stacks. In addition, we constructed a way for an attacker to locate said stacks without relying on guessing. In particular, we examined in detail the *Thread Control Block (TCB)* and *Thread Local Storage (TLS)*.

Whenever the system spawns a new thread for a program, it also initializes a corresponding Thread Control Block (TCB), which holds information about the thread (e.g., the address of its stack). However, once an attacker knows the location of the TCB, she also (already) has the stack location as the TCB is placed on the newly allocated stack of the thread. An exception is the creation of the main thread where the TCB is allocated in a memory region that has been mapped, with `mmap()`, during program initialization. Since the initialization of the program startup is deterministic, the TCB of the main stack is located at a fixed offset from the base address of `mmap()` (which can be easily inferred by leaked pointers into libraries).

Moreover, obtaining the address of the TCB is often easy, as a reference to it is stored in memory and passed to functions of the `pthread` library. While not visible to the programmer, the metadata required to manage threads in multi-threaded applications can also leak the address of the thread stacks. If an attacker is able to obtain this management data, she is also able to infer the location of stacks. Note that the management data is stored in the TCB because threads allocate their own stacks, so they need to free them as well. Furthermore, we found that the TCB also contains a pointer to a linked list with all TCBs for a process, so all stacks can be leaked this way.

Additionally, TLS consists of a static portion and a dynamic portion and the system happens to allocate the static portion of the TLS directly next to the TCB. The TLS portions are managed through the Dynamic Thread Vector (DTV) structure which is allocated on the heap at thread initialization and pointed to by the TCB. Leaking the location of DTV will also reveal the stack location.

Another way to obtain the location of the stacks is using global variables in `libpthread.so`. The locations of active stacks are saved in a double linked list called `stacks_used` which can be accessed if the location of the data section of `libpthread` is known to an attacker.

In summary, our analysis of the implementation reveals that references to sensitive information (in our case safe stacks) do occur in unexpected places in practice. While these issues may not be fundamental, given the complexity of the environment and the operating system, delivering a sound implementation of IH-based defenses is challenging. All references *should* be accounted for in a production defense that regards stack locations as sensitive information. We even argue that any IH-hardening solution (like the one presented in this paper) *should* take implementation flaws of defense solutions such as CPI into account, since they are common and often not under direct control of the solution (e.g., because of external code and libraries).

4.2 Attacks with Thread Spraying

While prior research has already demonstrated that information hiding mechanisms which utilize a large safe area are vulnerable to brute-force attacks [17], our research question is: are *small* safe areas without references to them really more secure than large safe areas? More generally speaking, we explore the limitations of hiding information in an address space and discuss potential attacks and challenges.

In the following, we investigate in detail CPI’s SafeStack as an illustrative example. While the safe area itself is very large (dependent on the implementation it may have sizes of 2^{42} or $2^{30.4}$ [17, 25]), a safe stack is only a few MB in size and hence it is challenging to locate it in the huge address space. We analyze the SafeStack implementation available in the LLVM compiler toolchain. As discussed above, the safe stack keeps only safe variables and return addresses, while unsafe variables are moved to an unsafe stack. Hence, an attacker—who has the possibility to read and write arbitrary memory—still cannot leak contents of the safe stack and cannot overwrite return addresses: she needs to locate the safe stack first.

We study if such an attack is feasible against web browsers, given the fact that they represent one of the most prominent attack targets. We thus compiled and

linked Mozilla Firefox (version 38.0.5) for Linux using the `-fsanitize=safe-stack` flag of the clang compiler and verified that SafeStack is enabled during runtime. We observed that safe stacks are normally relatively small: each thread gets its own safe stack, which is between 2MB (2^{21} bytes; 2^9 pages) and 8MB (2^{23} bytes; 2^{11} pages) in size. With 28 bits of entropy in the 64-bit Linux ASLR implementation, there are 2^{28} possible page-aligned start addresses for a stack. Hence, an adversary needs at least 2^{19} probes to locate a 2MB stack when sweeping through memory in a brute-force manner. In practice, such an attack seems to be infeasible. For server applications, a brute-force attack would be detectable by external means as it leads to many observable crashes [25].

However, an attacker might succeed with the following strategy to reduce the randomization entropy: while it is hard to find a *single* instance of a safe stack inside a large address space, the task is much easier if she can force the program to generate a lot of safe stacks with a certain structure and then locate just *one* of them. Thus, from a high-level perspective our attack forces the program to generate a large number of safe stacks, a technique we call *thread spraying*. Once the address space is populated with many stacks, we make sure that each stack has a certain structure that helps us to locate an individual stack within the address space. For this, we make use of a technique that we term *stack spraying*, to spray each stack in such a way that we can later easily recognize it. Finally, via a brute-force search, we can then scan the address space and locate a safe stack in a few seconds. In the following, we describe each step in more detail.

4.2.1 Thread Spraying

Our basic insight is that an adversary can abuse legitimate functions to create new stacks, and thereby decrease the entropy. Below, we explain how we performed the thread spraying step in our attack on Firefox. Furthermore, we show that the thread spraying technique is also possible in other applications, namely Chrome and MySQL.

Thread Spraying in Firefox: Our thread spraying in Firefox is based on the observation that an attacker within JavaScript can start *web workers* and each web worker is represented as a stand-alone thread. Thus, the more web workers we start, the more safe stacks are created and the more the randomization entropy drops. Thread spraying may spawn a huge number of threads. In empirical tests on Firefox we were able to spawn up to 30,000 web workers, which leads to 30,000 stacks with a size of 2MB each that populate the address space. In our attack, we implemented this with a malicious website that consists of 1,500 iframes. Each iframe, loading

a webpage from distinct domain name, allows the creation of 20 web workers. As we will show later, forcing the creation of 2,000 or even only 200 stacks is enough in practical settings to locate one of the safe stacks reliably. Fortunately, launching this lower number of concurrent threads is much less resource intensive and the performance impact is small.

Thread Spraying in Chrome: We also tested if Google Chrome (version 45.0.2454.93) is prone to thread spraying and found that Chrome only allows around 60 worker threads in the standard configuration. An investigation revealed that this number is constrained by the total amount of memory that can be allocated for worker threads. When we request more worker threads, the Chrome process aborts as it is unable to allocate memory for the newly requested thread. However, if the attacker has a write primitive, she can perform a data corruption attack [12] and modify a variable that has an effect on the size of the memory space being allocated for worker threads. In Chrome, we found that when we decrease the value of the *global* data variable `g_lazy_virtual_memory`, Chrome will allocate less memory space for a worker thread. The less space allocated, the more worker threads it can spawn. As a result, we were able to spawn up to 250 worker threads, with a default stack size of 8MB, after locating and modifying this data variable, during runtime, in the `bss` section of the Chrome binary.

Thread Spraying in MySQL: We also evaluated the thread spraying attack on the popular MySQL database server (version 5.1.65). Interestingly, MySQL creates a new thread for each new client connection. By default, the maximum number of simultaneous connections is 151 and each thread is created with a stacksize of 256KB. With 151 threads, this amounts to 37.8MB of safe stack area in the memory space which corresponds to spawning just ~19 Firefox or ~5 Chrome worker threads. This would make it hard to perform a successful thread spraying attack. However, as in the Chrome use case above, an attacker with a write primitive can corrupt exactly those variables that constrain the number of threads—using a data-oriented attack [12]. We found that the number of threads in MySQL is constrained by the global variables `max_connections` and `alarm_queue`. Increasing them, allows an attacker to create more connections and thus more threads. Since MySQL has a default timeout of 10 seconds for connections, it may be hard to keep a high number of threads alive simultaneously, but it is just as easy to overwrite the global variables `connect_timeout` and `connection_attrib`, which contains the stack size used when creating a thread for a new client connection. In a simple test we were able to create more than 1000 threads with a stacksize of 8MB.

Protecting the Thread Limits: In some applications, such as Chrome and MySQL, there are global variables that are associated explicitly or implicitly with thread creation. For example, in Chrome there is `g_lazy_virtual_memory` which, if reduced, allows for the creation of more worker threads. Placing these variables in read-only memory can potentially mitigate the thread-spraying attacks, however, it is unclear if the application’s behavior is also affected. In Section 5 we present a defense system that protects applications from all attacks discussed in this section without relying on protecting limits associated with thread creation.

4.2.2 Stack Spraying

At this point, we forced the creation of many stacks and thus the address space contains many copies of safe stacks. Next, we prepare each stack such that it contains a signature that helps us to recognize a stack later. This is necessary since we scan in the next step the memory space and look for these signatures in order to confirm that we have indeed found a safe stack (with high probability). In analogy with our first phase, we term this technique *stack spraying*.

From a technical point of view, we realize stack spraying in our attack as follows. Recall that a safe stack assumes that certain variables are safe and this is the case for basic data types such as integers or double-precision floating point values. Moreover, Firefox stores double-precision values in their hexadecimal form in memory. For instance, the number $2.261634509803921 * 10^6$ is stored as `0x4141414141414140` in memory. Additionally, calling a JavaScript function with a double-precision float value as parameter leads to the placement of this value on the safe stack since the system considers it safe. We exploit this feature to (i) fill the stack with values we can recognize and (ii) occupy as much stack space as possible. We therefore use a recursive JavaScript function in every worker which takes a large number of parameters. We call this function recursively until the JavaScript interpreter throws a JavaScript Error (*too much recursion*). As we can catch the error within JavaScript, we create as many stack frames as possible and keep the occupied stack space alive. Of course, other implementations of stack spraying are also possible.

A thread’s initial stack space contains various safe variables and return addresses before we call the recursive function the first time. Thus, this space is not controllable, but its size does not vary significantly across different program runs. For example, in our tests the initially occupied stack space had a size of approximately three memory pages (0x3000 bytes) in each worker. A sprayed stack frame consists of the values that the recursive function retrieves as parameters and is additionally

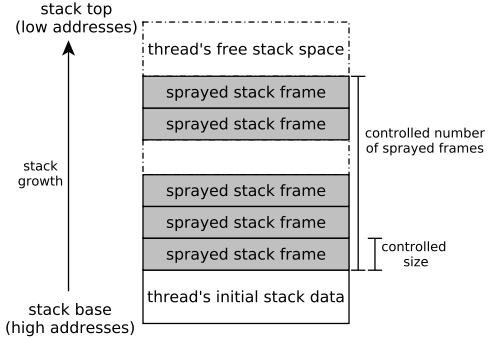


Figure 1: Memory layout of Firefox with CPI’s SafeStack filled with sprayed stack frames

interspersed with data intended to reside in a stack frame. As the size of this data is predictable, we can control the size of the stack frame with the number of parameters passed to the recursive function. While the number of sprayed stack frames is controllable via the number of recursive calls, we perform as many recursive calls as the JavaScript interpreter allows.

Figure 1 illustrates the memory layout of a sprayed safe stack after the first two phases of our attack. Since the system keeps safe variables such as double-precision floating point values on the safe stack, the memory can be filled with controlled stack frames which contain user-controlled signatures in a controllable number. Thus, we generate a repetitive pattern on the safe stacks of web workers, which leads to the following observations:

- The probability of hitting a stack during memory probing attempts increases, as the allocated space of a safe stack is filled with a recognizable signature.
- Where safe stacks are not consecutive to each other, they are separated only by small memory regions. Thus, probing with stack-sized steps is possible which reduces the number of probes even further.
- On a signature hit, we can safely read in sprayed frame sized steps towards higher addresses until we do not hit a signature anymore. This tells us that we have reached the beginning of the thread’s stack, which we can disclose further to manipulate a return address.

4.3 Scanning methodologies

During our experiments we developed multiple scanning methods fitted to different defense scenarios. In the following we shortly describe the observations leading to the development of each and evaluate them against the targeted defense measures. The first two techniques are targeted at the standard ASLR while the last technique

is also successful against an improved version. For our evaluation we assumed that an attacker can not always rely on the stacks being located close to each other. As such we implemented a simple module that can be loaded via `LD_PRELOAD` and forces each call to `mmap`, associated with a stack creation (i.e. `MAP_STACK` provided in the flags argument), to allocate memory at a random address. This means every page is a potential stack base and our first two methods are no longer effective.

4.3.1 Naïve attack on adjacent stacks

The simplest attack is based on the observation that all stacks are allocated close to each other, starting from a randomly chosen base address. To investigate this observation, we spawned 200 worker threads and performed stack spray in each one. We chose a number of parameters for the recursive function such that each sprayed stack frame had a size of one page (4096 bytes). As each thread gets a stack of 2MB in size and individual stacks are grouped closely in memory, we can treat the compound of stacks as a coherent region of approximately 2^{28} B in size. To locate a safe stack we scan towards lower addresses with 2^{28} B sized steps starting at `0x7fffffff000`, the highest possible stack base. As soon as we hit mapped memory we start searching in page sized steps for our sprayed signature. We performed this scan on three Firefox runs and needed only 16755.0 probing attempts on average (consisting of 1241.3 2^{28} B sized probes and 15513.7 page sized probes) to locate a signature and thus a safe stack. While this method is simple to implement and locates stacks with a high probability it has a chance to miss a stack region, if our single probe of a potential stack region hits the gap between two stacks by chance. While retrying with a different starting offset is possible, the next method is more fit for this purpose.

4.3.2 Optimized attack on adjacent stacks

As a variation of our previous method we modified the scanning strategy based on our observations. During three runs with Firefox, we first launched 2,000 worker threads and again performed stack spraying in each of them. Afterwards we conducted our *memory scanning*. The results are shown in Table 1. As our *memory range*, we chose `0x7FxxxxYYYYY0`, whereby the least three significant bytes are fixed (`YYYYY0`) while the fourth and fifth byte remain variable (`xxxx`). This yields a memory range of $2^{16} = 65,536$ addresses: due to 28-bit entropy for top down stack allocations, each of the chosen ranges constitutes a *potential range* for stack allocations. The probability that one of the chosen ranges is *not* a potential stack range is negligibly small.

Table 1: Memory scans in Firefox on eight different ranges after thread and stack spraying was applied- In each range, byte four and five are variable (denoted by ****). Thus, each range consists of $2^{16} = 65536$ addresses. *Mapped* denotes the number of readable addresses, *S-hits* the number of addresses belonging to the safe stack that contain our signature, and *Non S-Hits* represent safe stack addresses not containing our signature. *False S-hits* means that our signature was found at an address not being part of a safe stack.

Memory Range	Run 1				Run 2				Run 3			
	Mapped	S-Hits	Non S-Hits	False S-Hits	Mapped	S-Hits	Non S-Hits	False S-Hits	Mapped	S-Hits	Non S-Hits	False S-Hits
0x7f****000000	878	184	95	0	886	122	138	0	480	73	134	0
0x7f****202020	886	198	74	0	889	154	125	0	482	104	127	0
0x7f****404040	884	182	98	0	890	122	139	1	482	71	129	0
0x7f****606060	890	197	66	0	887	152	123	0	485	107	136	0
0x7f****808080	889	182	92	0	891	123	136	0	482	70	135	0
0x7f****a0a0a0	889	193	60	0	891	152	126	2	482	105	140	0
0x7f****c0c0c0	888	190	86	2	893	122	139	0	485	73	138	0
0x7f****e0e0e0	892	195	64	2	889	151	123	1	485	101	142	1

On average, 753.1 addresses out of 65,536 were mapped for each scan range. The ranges we tested were all potential ranges for safe stacks. 138.5 times a signature was hit, meaning we hit an address being part of a safe stack (*S-hits*). 0.4 times a signature was hit which did not belong to a safe stack. That means we hit a false positive (*false S-hits*). These false hits occur due to the signature being written by the program to non-stack regions which reside in potential stack ranges.

Choosing the three least significant bytes as a fixed value has the advantage of greatly reducing the search space: instead of probing in 2MB steps—which would be necessary to sweep the potential stack locations without a chance of missing a stack—we exploit the observation that the distance between allocated pages varies. Taking this into account leads to the conclusion that stacks are distributed in a way that gives any value of these ranges a roughly equal chance of being part of one stack. While it is not guaranteed to get a hit with this scanning approach, we cover a bigger area in less time. Additionally, in case the first run did not turn up any stack, we are free to retry the scan using a different range. With an increasing number of retries we get close to a full coverage of the scan region, but at the same time we spend less time probing a region without any stacks.

4.3.3 Locating non-adjacent stacks

With our modifications to stack ASLR the methods presented so far have a high chance of missing a stack, because they probe a memory region several times larger than a single stack. Therefore we need to assume no relation between stack locations and are forced to scan for memory of the size of a single stack. With the randomization applied we split the memory into $C = 2^{47}/2^{21} = 2^{26}$ chunks, each representing a possible 2MB stack location. We ignore the fact that some locations are already occupied by modules and heaps, as we are able to distinguish this data from stack data. Also building a complete memory map and then skipping these regions, if possible at all, would take more time than checking for a false stack hit. Without thread spraying we would be forced to

locate a single stack, which would mean we would on average need 2^{25} probes. Even with a high scanning speed this would not be feasible. However by spawning more threads we can reduce the number of probes in practice significantly.

We tested two strategies for locating these stacks. In theory every location in the address space has an equal chance of containing a stack, so scanning with a linear sweep with a step size of one stack seems like a valid approach that allows for locating all stacks eventually. However we noticed that the amount of probes required to locate any stack significantly differed from the expected amount. This can be explained by big modules being loaded at addresses our sweep reaches before any stacks. Due to this mechanic we risk sampling the same module multiple times instead of moving on to a possible stack location. As such we employed a different strategy based on a purely random selection of addresses to probe. In total we performed nine measurements and were able to locate a stack with 33,462 probes on average.

4.3.4 Crash-Resistant Memory Scanning

To verify that an attacker can indeed locate CPI’s SafeStack when conducting a memory corruption exploit against Firefox, we combined thread and stack spraying with a crash-resistant primitive introduced by recent research [18]. Crash-resistant primitives rely on probing memory while surviving or not causing at all application crashes. Using a crash-resistant primitive, it is possible to probe entire memory regions for mapped pages from within JavaScript and either receive the bytes located at the specified address or a value indicating that the access failed. In case an access violation happens, then the event is handled by an exception handler provided by the application, which eventually survives the crash. An equivalent approach is probing memory using system calls that return an error without crashing when touching unmapped memory. Equipped with crash-resistant primitives, we are free to use any strategy to locate the safe stack without the risk of causing an application crash.

We choose to scan potential ranges which may include safe stacks and hence we choose one of the ranges shown in Table 1. To counteract false positives, we employ a heuristic based on the observation that thread and stack spraying yield stacks of a predetermined size, each of which contains a large number of addresses with our signature. Determining the stack size is easily done after any address inside a stack candidate is found, because we are free to probe higher and lower addresses to locate the border between the stack’s memory and neighboring unmapped memory. Once these boundaries are known, it is possible to use memory disclosures to apply the second part of our heuristic. This heuristic is implemented in `asm.js` as an optimization. As our code is nearly directly compiled to native code it is very fast to execute. Additionally, we mainly need to index an array, with its start address set to the lower boundary, which is a heavily optimized operation in `asm.js`. If the number of entries in this array matching our sprayed values is above a certain threshold, we conclude that the memory region is indeed a stack. A further optimization we chose was to scan from higher addresses to lower addresses: we observed that stacks are usually the memory regions with the highest address in a process, which means we most likely hit a stack first when scanning from the top.

With the overhead caused by the thread and stack spraying, we are not able to use the full speed offered by the probing primitive [18]. This results in an average probing time of 46s to locate a safe stack (including the time for thread and stack spraying). The speed decrease is mainly due to the fact that we need to keep the threads and stack frames alive. Our attack achieved this by simply entering an endless loop at the end, which leads to degraded performance. However as web workers are designed to handle computational intensive tasks in the background, the browser stays responsive, but the scanning speed is affected.

Tagging safe stacks with user controlled data is not the only option for locating a safe stack. As most free stack space is zeroed out, a simple heuristic can be used to scan for zeros instead of scanning for placed markers. The advantage is that shadow stacks which separate return addresses and data are still locatable. Another possibility is to scan for known return addresses near the base of the stack: as coarse-grained ASLR randomizes shared libraries on a per-module basis and libraries are page aligned, the last twelve bits of return addresses stay the same across program runs and remain recognizable.

Without the overhead caused by the thread and stack spraying, our scanning speed is increased to 16,500 probes per second. As our approximated scanning method requires 65,536 scans per run, we are able to finish one sweep in less than 4 seconds. However, this is only the worst case estimation when not hitting any

stack. As mentioned before, we are then free to retry using a different value. On average, we are able to locate a safe stack in 2.3 seconds during our empirical evaluation.

4.4 Discussion: Implications for ASLR-Guard

In the following, we discuss the potential of a similar attack against ASLR-Guard [26]. While this defense provides strong security guarantees, it might be vulnerable to a similar attack as the one demonstrated above: as the native stack is used for the AG-stack, we can locate it using our scanning method. If the randomization of AG-stack locations is left to the default ASLR implementation (i.e., the stacks receive the same amount of entropy and potential base addresses), we can use our existing approach and only need to adjust for the size of the stacks (if different) in addition to a different scanning heuristic. This results in a longer scanning time, but if recursion can again be forced by attacker-supplied code, the resulting AG-stack will also increase in size. Combined with the thread spraying attack, we are able to generate a large number of sizable stacks. The major difference is that we are not able to spray a chosen value on the AG-stack. Further research into dynamic code generation might allow for spraying specific byte sequences as return addresses, if code of the appropriate size can be generated. While we can not evaluate the security of ASLR-Guard since we do not have access to an implementation, it seems possible to locate the AG-stack and thus disclose unencrypted code addresses.

Besides the AG-stack, there are two additional memory regions that must be hidden from an attacker. First, the code regions of executable modules are moved to a random location, but they are still potentially readable. As the *mmap-wrapper* is used, they receive an entropy of 28 bits. Since the stacks also receive the same amount of entropy, a similar attack is possible. Scanning can be done in bigger steps if a large executable module is targeted. Second, a safe area called *safe vault* is used for the translation of encoded pointers and it needs to be protected. If either of those structures is located, an adversary is able to launch a code-reuse attack. However, she would be limited to attack types that do not require access to the stack (e.g., COOP [32]). As stated in the paper, an attacker has a chance of 1 in 2^{14} to hit any of these regions with a random memory probe. This results in the possibility of exhausting the search space in roughly one second with the memory probing primitive discussed earlier. Additional steps need to be taken in order to determine the specific region hit, though. This can include signatures for code sections or heuristics to identify the safe vault.

5 Reducing the Odds for Attackers

We developed a mechanism called Authenticating Page Mapper (APM), which hinders attacks probing for the safe areas. Our mechanism is based on a user-level page fault handler authenticating accesses to inactive pages in the safe area and, when possible, also artificially inflating the virtual memory region backing the safe area.

The first strategy seeks to limit the attack surface to active safe area pages in the working set of the application. Since the working set is normally much smaller than the virtual memory size (especially for modern defenses relying on safe areas with sparse virtual memory layouts [13,24]), this approach significantly increases the entropy for the attacker. Also, since a working set is normally stable for real applications [38], the steady-state performance of APM is negligible.

The second strategy ensures an increase in the number of inactive pages not in use by the application, serving as a large trip hazard surface to detect safe area probing attacks with very high probability. In addition, since we randomize the concrete placement of the safe area within the larger inflated virtual memory space, this mitigates the impact of implementation bugs that allow an attacker to disclose the virtual (but not the physical) memory space assigned to the inflated area. Finally, this ensures that, even an attacker attempting to stress-test an application and saturate the working set of a safe area is still exposed to a very large detection surface.

Notice that deterministic isolation, and not hiding, can secure any safe area if properly applied. However, isolation in 64-bit systems has not, yet, been broadly adopted, while CPI’s SafeStack is already available in the official LLVM tool-chain [2] and there are discussions for porting it to GCC [3], as well. We therefore seek for a system that rather hardens IH, than fully protects it. To that end, APM stands as a solution until a proper replacement of IH is adopted by current defenses.

5.1 Authenticating Accesses

To authenticate accesses, APM needs to interpose on all the page faults in the safe area. Page faults are normally handled by the OS, but due to the proliferation of virtualization and the need for live migration of virtual machines, new features that enable reliable page fault handling in user space have been incorporated in the Linux kernel [6]. We rely on such features to gain control when an instruction accesses one of the safe areas to authenticate it. To authenticate the access, we rely on unforgeable execution capabilities, such as the faulting instruction pointer and stack pointer, exported by the kernel to our page fault handler and thus trusted in our threat model (arbitrary memory read/write primitives in userland). Our design draws inspiration from re-

cent hardware solutions based on instruction pointer capabilities [37], but generalizes such solutions to generic *execution capabilities* and enforces them in software (in a probabilistic but efficient fashion) to harden IH-based solutions. An alternative option is to use `SIGSEGV` handlers, but this introduces compatibility problems, since applications may have their own `SIGSEGV` handler, faults can happen inside the kernel, etc. On the other hand, `userfaultfd` [6] is a fully integrated technique for reliable page fault handling in user space for Linux.

APM is implemented as a shared library on Linux and can be incorporated in programs protected by CPI, ASLR-Guard, or any other IH-based solution by preloading it at startup (e.g., through the `LD_PRELOAD` environment variable). Upon load, we notify the kernel that we wish to register a user-level page-fault handler for each of the process’s safe areas (i.e., using the `userfaultfd` and `ioctl` system calls).

When any of the safe area pages are first accessed through a read or write operation, the kernel invokes the corresponding handler we previously registered. The handler obtains the current instruction pointer (RIP on x86-64), the stack pointer, the stack base, and the faulting memory address from the kernel, and uses this information to authenticate the access. Authentication is performed according to defense-specific authentication rules. If the memory access fails authentication, the process is terminated. In the other cases, the handler signals the kernel to successfully map a new zero page into the virtual memory address which caused the page fault.

To support CPI and SafeStack in our current implementation, we interpose on calls to `mmap()` and `pthread_create()`. In particular, we intercept calls to `mmap()` to learn CPI’s safe area. This is easily accomplished because the safe area is 4TB and it is the only such mapping that will be made. Furthermore, we intercept `pthread_create()`, which is used to initialize thread-related structures and start a thread, to obtain the address and size of the safe stack allocated for the new thread. In the following subsections, we detail how we implement authentication rules for CPI and SafeStack using our execution capabilities.

5.2 CPI’s Authentication Rules

To access a safe area without storing its addresses in data memory, CPI (and other IH-based solutions) store its base address in a CPU register not in use by the application. However, as the number of CPU registers is limited, CPI relies on the segmentation register `gs` available on x86-64 architectures to store the base address. The CPI instrumentation simply accesses the safe area via an offset from that register. Listing 1 shows an example of a safe area-accessing instruction generated by CPI.

```
mov    %gs:0x10(%rax),%rcx
```

Listing 1: x86-64 code generated by CPI to read a value from the safe area.

Since `gs` is not used for any other purpose (it was selected for this reason) and the instrumentation is assumed to be trusted, APM authenticates accesses to the CPI safe area by verifying that the *instruction pointer* points to an instruction using the `gs` register. Therefore, since the attacker needs to access the safe area before actually gaining control of the vulnerable program, only legitimate instructions part of the instrumentation can be successfully authenticated by APM.

5.3 SafeStack’s Authentication Rules

Similar to CPI’s safe area, SafeStack’s primary stack (safe stack) is also accessed through a dedicated register (RSP on x86-64 architectures) which originally points to the top of the stack. When new values need to be pushed to it, e.g., due to a function call, the program allocates space by subtracting the number of bytes needed from RSP. This occurs explicitly or implicitly through the `call` instruction. Hence, to authenticate safe stack accesses, APM relies on the *stack pointer* (RSP) to verify the faulting instruction accesses only the allocated part of the stack. The latter extends from the current value of RSP to the base of the safe stack of each thread. We also need to allow accesses the red zone on x86-64.

5.4 Inflating the Safe Area

We inflate safe areas by allocating more virtual address space than it is needed for the area. For example, when a new safe stack is allocated, we can request 10 times the size the application needs. The effect of this inflation is that a larger part of the address space becomes “eligible” for memory-access authentication, amplifying our detection surface. Inflation is lightweight, since the kernel only allocates pages and page-table entries after a page is first accessed.

We implement our inflation strategy for SafeStack (CPI’s safe region is naturally “inflated” given the full memory shadowing approach used). To inflate thread stacks, our `pthread_create()` wrapper sets the stack size to a higher value (using `pthread_attr_setstacksize()`). For the main stack, initialized by the kernel, we implement inflation by increasing the stack size (using `setrlimit()`) before the application begins executing. Similar to CPI, we rely on the `mmap()`’s `MAP_NORESERVE` flag to avoid overcommitting a large amount of virtual memory in typical production settings.

To randomize the placement of each safe stack within the inflated virtual memory area, we randomize the initial value of RSP (moving it upward into the inflated area) while preserving the base address of the stack and the TCB in place. Since the base address of each stack is saved in memory (as we describe in Section 4), a memory leak can exfiltrate its base address. Our randomization strategy can mitigate such leaks by moving the safe stack working set to a random offset from the base address and exposing guided probing attacks to a large trip hazard surface in between.

6 Evaluation

In this section, we report on experimental results of our APM prototype. We evaluate the our solution in terms of performance, entropy gains (reducing the likelihood attackers will hit the target region), and detection guarantees provided by APM coped with our inflation strategy (authenticating memory accesses to the target region and raising alerts).

We performed our experiments on an HP Z230 machine with an intel i7-4770 CPU 3.40GHz and running Ubuntu 14.04.3 LTS and Linux kernel v4.3. Unless otherwise noted, we configured APM with the default inflation factor of 10x. We repeated all our experiments 5 times and report the median (with little variations across runs).

Performance To evaluate the APM’s performance we run the SPEC2006 suite, which includes benchmarks with very different memory access patterns. For each benchmark, we prepared three versions: (1) the original benchmark, denoted as *BL* (Baseline), (2) the benchmark compiled with CPI’s SafeStack only, denoted as *SS*, and (3) the benchmark compiled with full CPI support, denoted as *CPI*. Table 2 presents our results. Note that `perlbench` and `povray` fail to run when compiled with CPI, as also reported by other researchers [17]. Therefore, results for these particular cases are excluded from the table.

Not surprisingly, the overhead imposed by APM in all benchmarks and for all configurations (i.e., either compiled using SafeStack or full CPI) is very low. The geometric mean performance overhead increase is only 0.3% for *BL+APM*, 0.0% for *SS+APM* and 1.4% *CPI+APM*.

To confirm our performance results, we evaluated the APM-induced overhead on Chrome (version 45.0.2454.93) and Firefox (version 38.0.5) by running popular browser benchmarks—also used in prior work in the area [21, 23]—i.e., sunspider, octane, kraken, html5, balls and linelayout. Across all the benchmarks, we observed essentially no overhead (0.01% and 0.56% ge-

Apps	BL	BL + APM	SS	SS + APM	CPI	CPI + APM
astar	133.8 sec	1.004x	1.003x	1.002x	0.971x	0.985x
bzip2	82.6 sec	1.003x	1.002x	1.008x	1.039x	1.055x
dealII	229.4 sec	1.008x	1.009x	1.013x	0.887x	0.897x
gcc	19.2 sec	0.978x	0.982x	0.988x	1.368x	1.440x
gobmk	53.6 sec	1.001x	1.020x	1.018x	1.046x	1.046x
h264ref	51.6 sec	1.000x	1.009x	1.013x	1.028x	1.031x
hmmer	113.7 sec	0.996x	1.001x	0.996x	1.063x	1.066x
lbm	248.5 sec	1.002x	1.001x	1.002x	1.154x	1.159x
libquantum	274.1 sec	1.004x	1.015x	1.013x	1.231x	1.227x
mcf	237.4 sec	1.031x	0.998x	0.989x	1.046x	1.045x
milc	349.0 sec	1.009x	0.991x	0.997x	1.012x	1.023x
namd	306.8 sec	1.000x	1.001x	0.997x	1.031x	1.030x
omnetpp	358.8 sec	0.994x	1.017x	1.044x	1.377x	1.472x
perlbench	263.9 sec	1.004x	1.084x	1.091x	—	—
povray	121.3 sec	1.005x	1.092x	1.093x	—	—
sjeng	397.8 sec	1.004x	1.047x	1.051x	1.033x	1.031x
soplex	136.0 sec	1.001x	1.000x	0.951x	1.000x	0.997x
sphinx3	410.6 sec	0.987x	0.997x	0.995x	1.149x	1.138x
xalancbmk	189.0 sec	1.020x	1.042x	1.055x	1.679x	1.782x
<i>geo-mean</i>		1.003x	1.016x	1.016x	1.111x	1.125x

Table 2: SPEC CPU 2006 benchmark results. We present the overhead of hardening state-of-the art defenses with APM. BL and SS refer to *baseline* and *safe stack* (respectively), and CPI refers to CPI’s safe area.

ometric mean increases on Chrome and Firefox, respectively). These results confirm that, while APM may introduce a few expensive page faults early in the execution, once the working set of the running programs is fully loaded in memory, the steady-state performance overhead is close to zero. We believe this property makes APM an excellent candidate to immediately replace traditional information hiding on today’s production platforms.

Entropy Gains With APM in place, it becomes significantly harder for an adversary to locate a safe area hidden in the virtual address space. To quantify the entropy gains with APM in place, we ran again the SPEC2006 benchmarks in three different configurations, including a parallel shadow stack [14] other than SafeStack and full CPI. We present results for a parallel shadow stack to generalize our results to arbitrary shadow stack implementations in terms of entropy gains. A parallel shadow stack is an ideal candidate for generalization, since its shadow memory-based implementation consumes as much physical memory as a regular stack, thereby providing a worst-case scenario for our entropy gain guarantees.

For each configuration, we evaluated the entropy with and without APM in place and report the resulting gains. The entropy gain is computed as $\log_2(VMM/PMM)$, where VMM is the Virtual Mapped Memory size (in pages) and PMM is the Physical Mapped Memory size (in pages). To mimic a worst-case scenario for our en-

tropy gains, we measured PMM at the very end of our benchmarks, when the program has accessed as much memory as possible resulting in the largest resident set (and lowest entropy gains). Table 3 presents our results. Once again, as also reported by other researchers [17], perlbench and povray are excluded from the CPI configuration.

As expected, our results in Table 3 show that lower stack usage (i.e., lower PMM) results in higher entropy gains. Even more importantly, the entropy gains for CPI-enabled applications are substantial. In detail, we gain 11 bits of entropy even in the worst case (i.e., xalancbmk). In other cases, (e.g., bzip2) the entropy gains go up to 28 bits of entropy.

We find our experimental results extremely encouraging, given that, without essentially adding overhead to CPI’s fastest (but low-entropy) implementation, our techniques can provide better entropy than the slowest (probabilistic) CPI implementation [25]. SafeStack’s entropy gains are, as expected, significantly lower than CPI’s, but generally (only) slightly higher than a parallel shadow stack. In both cases, the entropy gains greatly vary across programs, ranging between 2 and 11 bits of entropy. This is due to the very different memory access patterns exhibited by different programs. Nevertheless, our strategy is always effective in nontrivially increasing the entropy for a marginal impact, providing a practical and immediate improvement for information hiding-protected applications in production.

Apps	Parallel Shadow Stack				SafeStack				CPI's (safe region)			
	VMM	PMM	EG	DG	VMM	PMM	EG	DG	VMM	PMM	EG	DG
astar	2048	3	> 9 bits	99.99 %	2048	2	10 bits	99.99 %	1 GP	201668	> 12 bits	99.98 %
bzip2	2048	4	9 bits	99.98 %	2048	1	11 bits	100.00 %	1 GP	4	28 bits	100.00 %
gcc	2048	112	> 4 bits	99.45 %	2048	8	8 bits	99.96 %	1 GP	121314	> 13 bits	99.99 %
gobmk	2048	27	> 6 bits	99.87 %	2048	7	> 8 bits	99.97 %	1 GP	5813	> 17 bits	100.00 %
h264ref	2048	5	> 8 bits	99.98 %	2048	2	10 bits	99.99 %	1 GP	6994	> 17 bits	100.00 %
hmmer	2048	3	> 9 bits	99.99 %	2048	2	10 bits	99.99 %	1 GP	36616	> 14 bits	100.00 %
lbm	2048	2	10 bits	99.99 %	2048	1	11 bits	100.00 %	1 GP	2	> 29 bits	100.00 %
libquantum	2048	1	11 bits	100.00 %	2048	2	10 bits	99.99 %	1 GP	66911	> 13 bits	99.99 %
mcf	2048	2	10 bits	99.99 %	2048	2	10 bits	99.99 %	1 GP	5107	> 17 bits	100.00 %
milc	2048	2	10 bits	99.99 %	2048	1	11 bits	100.00 %	1 GP	20017	> 15 bits	100.00 %
namd	2048	10	> 7 bits	99.95 %	2048	2	10 bits	99.99 %	1 GP	109	> 23 bits	100.00 %
omnetpp	2048	34	> 5 bits	99.83 %	2048	10	> 7 bits	99.95 %	1 GP	171316	> 12 bits	99.98 %
perlbench	2048	491	> 2 bits	97.60 %	2048	446	> 2 bits	97.82 %	—	—	—	—
povray	2048	7	> 8 bits	99.97 %	2048	4	9 bits	99.98 %	—	—	—	—
sjeng	2048	132	> 3 bits	99.36 %	2048	26	> 6 bits	99.87 %	1 GP	2	> 29 bits	100.00 %
soplex	2048	3	> 9 bits	99.99 %	2048	2	10 bits	99.99 %	1 GP	31673	> 15 bits	100.00 %
sphinx3	2048	12	> 7 bits	99.94 %	2048	2	10 bits	99.99 %	1 GP	11334	> 16 bits	100.00 %
xalancbmk	2048	496	> 2 bits	97.58 %	2048	494	> 2 bits	97.59 %	1 GP	316838	> 11 bits	99.97 %

Table 3: Entropy gains with our defense. VMM, PMM, EG, and DG refer to *Virtual Mapped Memory*, *Physical Mapped Memory*, *Entropy Gains* and *Detection Guarantees* (respectively). VMM and PMM are measured in number of pages. EG is given by $\log_2(VMM/PMM)$. DG is given by $(1 - PMM/(VMM * \text{inflation_factor})) * 100$, where the *inflation_factor* is set to default 10x for stacks and 1x for the already huge CPI's safe region. GP stands for *giga pages*, i.e., $1024 * 1024 * 1024$ regular pages. A regular page has a size of 4096 bytes.

Detection Guarantees Table 3 also illustrates the detection guarantees provided by APM when coped with the default 10x inflation strategy. The detection guarantees reflect the odds of an attacker being flagged probing into the inflated trip hazard area rather than in any of the safe pages mapped in physical memory. As shown in the table, APM offers very strong detection guarantees across all our configurations. Naturally, the detection guarantees are stronger as the size of the inflated trip hazard area (i.e., $VMM * \text{inflation_factor} - PMM$) increases compared to the resident size (i.e., PMM). The benefits are, again, even more evident for CPI's sparse and huge safe area, which registered 100% detection guarantees in almost all cases. Even in the worst case (i.e., *xalancbmk*), CPI retains 316,838 trip hazard pages at the end of the benchmark, resulting in 99.97% detection guarantees.

To lower the odds of being detected, an attacker may attempt to force the program to allocate as many safe area physical pages as possible, naturally reducing the number of trip hazard pages. We consider the impact of this scenario in Firefox, with a JS-enabled attacker spraying the stack to bypass APM. Figure 2 presents our results for different inflation factors assuming an attacker able to spray only the JS-visible part of the stack (1MB) or the entire stack to its limit (2MB). As shown in the figure, in both cases, APM provides good detection guarantees for reasonable values of the inflation factor and up to 95% with a 20x inflation (full spraying setting). Even in our default configuration, with a 10x inflation, APM offers adequate detection guarantees in practice (90% for the full spraying setting).

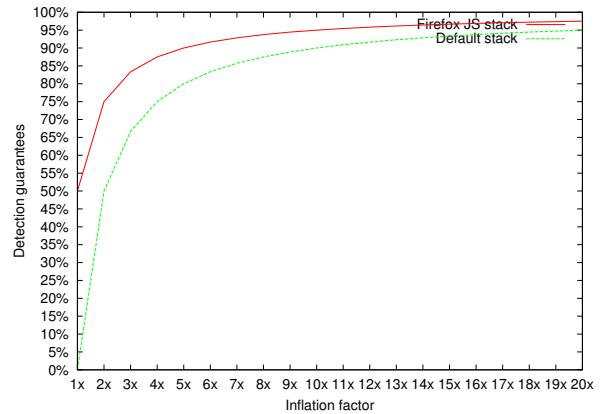


Figure 2: Effect of stack spraying (JS-visible or default stack) on our detection guarantees (DGs) across different inflation factors.

Limitations APM aims at hardening IH, but does not guarantee that a defense based on IH is fully protected against arbitrary attacks. Defenses that rely on IH should properly isolate the safe area to preserve the integrity and/or confidentiality of sensitive data. In the absence of strong (e.g., hardware-based) isolation, however, APM can transparently raise the bar for attackers, since it can offer protection without programs being aware of it (no re-compilation or binary instrumentation is needed). Nevertheless, certain attacks can still reduce the entropy and the detection guarantees provided by APM. For example, an attacker may be able to locate the base address of an inflated safe area by exploiting an implementation flaw or the recent allocation oracle side channel [28].

While the entropy is reduced, the trip hazard pages still deter guided probing attacks in the inflated area. However, if an implementation flaw or other side channels were to allow an attacker to leak a pointer to an active safe area page in use by the application (e.g., RSP), APM would no longer be able to detect the corresponding malicious access, since such page has already been authenticated by prior legitimate application accesses.

7 Conclusion

Information hiding is at the heart of some of the most sophisticated defenses against control-flow hijacking attacks. The assumption is that an attacker will not be able to locate a small number of pages tucked away at a random location in a huge address space if there are no references to these pages in memory. In this paper, we challenge this assumption and demonstrate that it is not always true for complex software systems such as Mozilla Firefox. More specifically, we examined CPI’s SafeStack since it is considered to be the state-of-the-art defense. In a first step, we analyzed the implementation and found that there were still several pointers to the hidden memory area in memory. An attacker can potentially abuse a single such pointer to bypass the defense. More seriously still, the protection offered by high entropy is undermined by thread spraying—a novel technique whereby the attacker causes the target program to spawn many threads in order to fill the address space with as many safe stacks as possible. Doing so reduces the entropy to the point that brute-force attacks become viable again. We demonstrated the practicality of thread spraying by way of an attack against Firefox, Chrome and MySQL, protected with CPI’s SafeStack.

To mitigate such entropy-reducing attacks, we propose an IH hardening strategy, namely APM. Based on a user-space page fault handler, APM allows accessing of pages on demand only and vets each first access to a currently guarded page. The additional protection provided by the page fault handler greatly improves the pseudo-isolation offered by information hiding, making it a concrete candidate to replace traditional information hiding in production until stronger (e.g., hardware-based) isolation techniques find practical applicability. Most notably, our approach can be used to harden existing defenses against control-flow hijacking attacks with barely measurable overhead.

Acknowledgements

We thank the reviewers for their valuable feedback. This work was supported by Netherlands Organisation for Scientific Research through the NWO 639.023.309

VICI “Dowsing” project, by the European Commission through project H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 64457, and by ONR through grant N00014-16-1-2261. Any opinions, findings, conclusions and recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US Government, or the ONR.

Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to the relevant parties.

References

- [1] Applications using older atl components may experience conflicts with dep. <https://support.microsoft.com/en-us/kb/948468>.
- [2] Clang’s SafeStack. <http://clang.llvm.org/docs/SafeStack.html>.
- [3] Discussion for porting SafeStack to GCC. <https://gcc.gnu.org/ml/gcc/2016-04/msg00083.html>.
- [4] ALTEKAR, G., BAGRACKI, I., BURSTEIN, P., AND SCHULTZ, A. Opus: online patches and updates for security. In *USENIX Security ’05*.
- [5] ANDERSEN, S., AND ABELLA, V. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention, 2004. <http://technet.microsoft.com/en-us/library/bb457155.aspx>.
- [6] ARCANGELII, A. Userfaultfd: handling userfaults from userland.
- [7] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security ’14*.
- [8] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *IEEE S&P ’14*.
- [9] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: A new class of code-reuse attack. In *ASIA CCS ’11*.
- [10] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE S&P ’16*.
- [11] CARLINI, N., AND WAGNER, D. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security ’14*.
- [12] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *USENIX Security ’05*.
- [13] DANG, T. H., MANIATIS, P., AND WAGNER, D. The performance cost of shadow stacks and stack canaries. In *ASIA CCS ’15*.
- [14] DANG, T. H., MANIATIS, P., AND WAGNER, D. The performance cost of shadow stacks and stack canaries. In *ASIA CCS ’15*.
- [15] DAVI, L., LIEBCHEN, C., SADEGHI, A. R., SNOW, K. Z., AND MONROSE, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS ’15*.

- [16] DAVI, L., SADEGHI, A.-R., LEHMANN, D., AND MONROE, F. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security '14*.
- [17] EVANS, I., FINGERET, S., GONZALEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., AND OKHRAVI, H. Missing the point(er): On the effectiveness of code pointer integrity. In *IEEE S&P '15*.
- [18] GAWLIK, R., KOLLENDA, B., KOPPE, P., GARMANY, B., AND HOLZ, T. Enabling client-side crash-resistance to overcome diversification and information hiding. In *NDSS '16*.
- [19] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security '12*.
- [20] GÖKTAŞ, E., ATHANASOPOULOS, E., POLYCHRONAKIS, M., BOS, H., AND PORTOKALIDIS, G. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *USENIX Security '14*.
- [21] HALLER, I., GÖKTAŞ, E., ATHANASOPOULOS, E., PORTOKALIDIS, G., AND BOS, H. Shrinkwrap: Vtable protection without loose ends. In *ACSAC '15*.
- [22] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In *IEEE S&P '13*.
- [23] JANG, D., TATLOCK, Z., AND LERNER, S. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *NDSS '14*.
- [24] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer Integrity. In *OSDI '14*.
- [25] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., AND SONG, D. Poster: Getting the point(er): On the feasibility of attacks on code-pointer integrity. In *IEEE S&P '15*.
- [26] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., AND LEE, W. Aslr-guard: Stopping address space leakage for code reuse attacks. In *CCS '15*.
- [27] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., AND FRANZ, M. Opaque Control-Flow Integrity. In *NDSS '15*.
- [28] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *USENIX Sec '16*.
- [29] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *S&P '12*.
- [30] PAX TEAM. Address Space Layout Randomization (ASLR), 2003. pax.grsecurity.net/docs/aslr.txt.
- [31] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under unix. In *USENIX ATC '95*.
- [32] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE S&P '15*.
- [33] SEIBERT, J., OKHRAVI, H., AND SÖDERSTRÖM, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *CCS '14*.
- [34] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS '07*.
- [35] SNOW, K. Z., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., MONROE, F., AND SADEGHI, A.-R. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE S&P '13*.
- [36] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIJSESENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption. In *EuroSec EWSS '09*.
- [37] VILANOVA, L., BEN-YEHUDA, M., NAVARRO, N., ETSION, Y., AND VALERO, M. Codoms: Protecting software with code-centric memory domains. In *ISCA '14*.
- [38] VOGT, D., MIRAGLIA, A., PORTOKALIDIS, G., BOS, H., TANENBAUM, A. S., AND GIUFFRIDA, C. Speculative memory checkpointing. In *Middleware '15*.
- [39] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SOSP '93*.
- [40] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *CCS '12*.
- [41] XI CHEN, A. S., DENNIS ANDRIESSE, H. B., AND GIUFFRIDA, C. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS '15*.
- [42] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORM, T., OKASAKA, S., NARULA, N., FULLAGAR, N., AND INC, G. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE S&P '09*.
- [43] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE S&P '13*.
- [44] ZHANG, M., AND SEKAR, R. Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-World ROP Attacks. In *ACSAC '15*.

Poking Holes in Information Hiding

Angelos Oikonomopoulos
Vrije Universiteit Amsterdam
a.oikonomopoulos@vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herberth@cs.vu.nl

Elias Athanasopoulos
Vrije Universiteit Amsterdam
i.a.athanasopoulos@vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Abstract

ASLR is no longer a strong defense in itself, but it still serves as a foundation for sophisticated defenses that use randomization for pseudo-isolation. Crucially, these defenses hide sensitive information (such as shadow stacks and safe regions) at a random position in a very large address space. Previous attacks on randomization-based information hiding rely on complicated side channels and/or probing of the mapped memory regions. Assuming no weaknesses exist in the implementation of hidden regions, the attacks typically lead to many crashes or other visible side-effects. For this reason, many researchers still consider the pseudo-isolation offered by ASLR sufficiently strong in practice.

We introduce powerful new primitives to show that this faith in ASLR-based information hiding is misplaced, and that attackers can break ASLR and find hidden regions on 32 bit and 64 bit Linux systems quickly with very few malicious inputs. Rather than building on memory accesses that probe the allocated memory areas, we determine the sizes of the *unallocated holes* in the address space by repeatedly allocating large chunks of memory. Given the sizes, an attacker can infer the location of the hidden region with few or no side-effects. We show that allocation oracles are pervasive and evaluate our primitives on real-world server applications.

1 Introduction

While Address Space Layout Randomization (ASLR) by itself no longer ranks as a strong defense against advanced attacks due to the abundance of memory disclosure bugs [1], it is still an essential foundation for more sophisticated defenses that use randomization to provide fast pseudo-isolation. Specifically, these defenses hide important sensitive information (such as shadow stacks [2], safe regions [3], or redirection tables [4]) at a random position in a very large address space. The un-

derlying and crucial assumption is that an attacker is not able to detect the location of the hidden regions.

Thus, the strength of all these defenses hinges entirely on the ASLR-provided obscurity of the hidden region. Our research question is whether such trust in the randomization schemes of modern systems like Linux is justified. In particular, we show that it is not, and introduce powerful new primitives, *allocation oracles*, that allow attackers to stealthily break ASLR on Linux and quickly find hidden regions on both 32-bit and 64-bit systems.

Randomization for information hiding Most operating systems today employ coarse-grained ASLR [5] which maps the different parts of the process (the stack, heap, and `mmap` region) in random locations in memory. The amount of randomness determines the strength of the defense. As an extreme example, the entropy for the `mmap` base on 32-bit Linux is as low as 8 bits, which means that the region can start at 256 possible locations in memory. This is well within range of a relatively stealthy brute-force attack. On 64-bit machines, however, the entropy of the `mmap` region on Linux is 28 bits and brute forcing is no longer considered practical. Unfortunately, whatever the granularity and entropy, address space randomization is vulnerable to information disclosure attacks. For example, in the absence of additional defenses and given a single code pointer, attackers can easily find other code pointers and eventually enough code to stitch together a code reuse attack [1].

However, powerful new defenses have evolved that still rely on randomization, but this time for the purpose of hiding a secret region of memory in a large address space [2, 3, 4]. Typically, they ensure the confidentiality and integrity of code pointers (such as return addresses, function pointers, and VTable pointers) [3, 6]. As manipulating a code pointer is vital for an attacker to take control of the program, preventing unauthorized access to code pointers also prevents such attacks. Thus, instead of storing code pointers in the program code, the heap, or

the stack, they place them in an *isolated* memory region. For instance, some defenses store the return addresses on an isolated “shadow” stack. Such defenses work as long as attackers cannot access the isolated region.

While it is possible to isolate these regions using techniques such as Software Fault Isolation (SFI) [7, 8], most existing solutions adopt cheaper ASLR-based *pseudo-isolation*—presumably for performance reasons or since commodity hardware-supported fault isolation can dramatically limit the size of the address space [9]. In other words, they resort to *information hiding* by placing the region at a random location in a very large virtual (and mostly inaccessible) address space and making sure that no pointers to it exist in regular memory.

The role of ASLR in information hiding is quite different to its use in countering code-reuse attacks directly, since even a strong *read* or *write* primitive ceases to be trivially sufficient for breaking the defense. Specifically, hiding all sensitive pointers forces attackers to probe the address space repeatedly (with the number of probes proportional to the size of the address space) and risk detection from crashes [10], or other observable events [11]. While Evans et al. [12] show that problematic implementations relying on huge hidden regions are still vulnerable to crashless probing attacks, more advanced defenses are not [6]. Indeed, the many new defenses that rely on information hiding show that ASLR is widely considered to offer strong isolation.

Allocation oracles Unlike previous approaches, our attack does not revolve around probing valid areas of allocated memory. Instead, we introduce new primitives to gauge the size of the *holes* in the address space. The key idea is that once an attacker knows the sizes of the holes, she can infer the start of the hidden regions. In other words, even if all the pointers into the hidden regions have been removed, the sizes of the unallocated parts of the address space “*point*” into the hidden regions.

To gauge the sizes of the holes, we introduce *allocation oracles*: information disclosure primitives that allow an attacker to allocate large chunks of memory repeatedly and thus probe for the possible sizes of the largest hole in the address space. In most cases, she can use binary search to find the exact size after a handful of probing attempts. The pre-conditions for allocation primitives are the ability to make repeated, arbitrarily large memory allocations, and to detect the success or failure of such allocation attempts. For instance, the simplest oracle might be the length field in a protocol header that controls the amount of memory a server allocates for a request [13]. More reliably, the attacker may corrupt a value in memory that is later used as an allocation size. Assuming the attacker can distinguish between success and failure of the allocations, this primitive operates as

an allocation oracle. We will show that such cases are common in real-world server programs.

Allocation oracles come in two main forms. *Ephemeral* allocation oracles perform allocations that have a short lifetime. For instance, a server which allocates memory for a client request and frees it after sending the reply. Ephemeral allocation oracles are the most effective in detecting the hidden regions. In the absence of ephemeral allocation oracles, we may find *persistent* allocation oracles. In this case, the allocation is permanent. This property alone makes attacks harder, but not impossible. In this paper, we present exploitation techniques and examples using either kind of oracle, as well as a powerful combination of the two. This combination allows an attacker to disclose the location of small hidden regions *arbitrarily located* in an *arbitrarily large* address space with *no crashes* or other detection-prone side effects.

Contributions We make the following contributions:

- We introduce new types of disclosure primitives, termed *allocation oracles*. Unlike existing primitives, allocation oracles do not work by accessing memory addresses, but instead probe the address space for “holes”. We describe primitives for both ephemeral and persistent allocations, and show how to combine them to break information hiding.
- We describe a methodology to assist an attacker in easily discovering both ephemeral and persistent allocation primitives in real programs. We show that such primitives are very common in practice. When real-world instances of our primitives are imperfect, we show how an attacker can exploit *timing side channels* to mount effective attacks.
- We show that our primitives can be exploited to mount end-to-end disclosure attacks on several real-world server programs. Our attacks render ASLR ineffective even on 64-bit (or larger) systems and show that an attacker can quickly locate hidden regions of existing defenses with little or no trace.

Organization We introduce the threat model in Section 2. Section 3 provides the necessary background for our attacks, presented in Section 4. We then describe our methodology for discovering memory allocation primitives (Section 5) and evaluate their availability and the effectiveness of the proposed attacks in Section 7. Finally, we discuss the implications for the defense mechanisms that rely on ASLR for information hiding (Section 7.6), consider mitigations (Section 8), place our attacks in the context of related work (Section 9), and draw conclusions in Section 10.

2 Threat model and assumptions

The attacks presented in this paper apply to programs that contain vulnerabilities, but are, nevertheless, protected using state-of-the-art defenses. The sensitive data, vital for the correct operation of the defense, is isolated in a hidden region by means of information hiding. Hardware-based isolation, realized with segmentation on 32 bit x86 architectures, is not available. These assumptions correspond to some of the most advanced anti-exploitation defenses for x86-64 today [3]. Note that we assume an *ideal* information hiding implementation, i.e., all sensitive information is in a hidden region at a truly random location in a large virtual address space and the code that performs this pseudo-isolation, as well as the defense itself, contain no faults. In addition, we assume that the separation of sensitive and non-sensitive data is perfect; the process memory holds no references to the hidden region, and following pointers from non-sensitive regions can never lead to pointers into the hidden region.

We further assume an attacker with arbitrary memory read and write primitives. In other words, the attacker can read or write any byte in the virtual address space. However, we consider that all sensitive data, which could allow an attacker change the control flow of the program in order to execute arbitrary code, is hidden in the hidden region. Therefore, although the attacker can read any byte in memory, she cannot probe the address space by brute force without incurring program crashes or other noticeable events with high probability.

We assume that the target application runs on a modern Linux system with memory overcommit. This is a common configuration in many production systems, either because of the pervasive use of virtualization technologies [14], or because this is required or explicitly recommended for popular and complex services, Redis [15] and Hadoop [16] among others. We also generally consider (real-world) applications that either handle allocation failures appropriately or do not crash in a way that triggers a re-randomization (e.g., by forking and using `execve` to replace the worker process image) when the allocation request cannot be serviced. The goal of the attacker is to carefully utilize memory oracles to poke holes into the information hiding and reveal the location of the hidden region.

3 Background

In this section, we illustrate the organization of a typical process' virtual memory address space. While most of the discussion is based on Linux-based operating systems, we present fairly generic address space organization principles which apply to other systems as well. Un-

Hole	Min	Max	Entropy ¹
A	130,044GiB	131,068GiB	28 bits
B	1GiB	1,028GiB	28 bits
C	4KiB	4GiB	20 bits

Table 1: Virtual memory hole ranges for a 64-bit position-independent executable (PIE) on Linux.

derstanding the memory layout of processes is vital for comprehending the mechanics of memory allocation oracles, detailed in the following sections.

The default address space of a typical x86_64 position-independent executable (PIE) on Linux (kernel version 3.14.7 used as a reference) is depicted in Figure 1. The system randomly selects an address which serves as the starting offset of the process' `mmap` space. In kernel concepts, this is a per-address-space `mmap_base` variable. Shared objects, including the PIE executable itself, are allocated in this virtual memory-mapped area, which extends towards lower addresses. Figure 1 also illustrates several *holes* (unmapped regions) fragmenting the address space. Such holes have different purposes and semantics.

To support typical dynamic memory allocations, the process relies on a separate [`heap`] space, at the lowest level managed by `brk/sbrk` calls. As the stack grows down on x86, the heap is naturally designed to grow up towards the stack. The size of the hole between these two regions is randomized. The stack, in turn, is located at a random offset from the end of the user address space (i.e., at `0x7fffffffffffff`), giving rise to another variable-sized hole at the top.

To protect against trivial exploitation of NULL pointer dereferences by the kernel [17], processes are not allowed to map or access addresses ranging from zero up to an administrator-configurable limit (i.e., `vm.mmap_min_addr`, which defaults to 64KiB). Additionally, the small hole between the stack and VDSO is typically less than 2MiB. In less than 1% of the invocations, the VDSO object will end up either adjacent to the stack or adjacent to the linker object. In both cases, the layout is effectively the same, except that the small random hole may not be present.

In practice, the uncertainty in the layout of the address space is dominated by the sizes of the large hole from `vm.mmap_min_addr` to the end of the `mmap` space (hereafter referred to as hole A), the hole between the stack and heap (named B) and the hole covering the top of the user address space (named C). While there may be holes between the loaded shared objects, those are normally of a known (fixed) size. The sizes of these holes are all uniformly distributed in the ranges shown in Table 1.

¹Calculated under the assumption that the distributions are indepen-

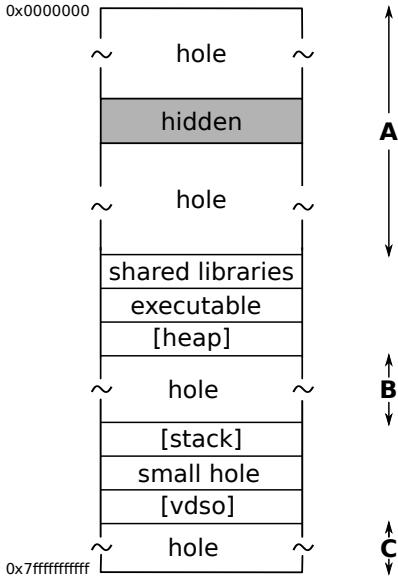


Figure 1: Virtual memory address space layout for a 64-bit position-independent executable (PIE) on Linux.

4 Memory Allocation Oracles

In this section, we thoroughly discuss the mechanics of two memory-allocation oracles, which can dramatically reduce the entropy of ASLR for accurately locating a hidden region in the virtual address space. The oracles can be realized through an *ephemeral allocation primitive* (EAP) and a *persistent allocation primitive* (PAP), respectively. Both primitives can be triggered by attacker-controlled input, say an HTTP request in a typical web server, and force a legitimate program path to allocate virtual memory with attacker-controlled size. By repeatedly using such primitives and monitoring the behavior of the target program (e.g., the error code in a HTTP response message), the attacker can infer the size of *holes* (unallocated space) in the virtual memory address space and learn key properties on its layout.

Whenever an EAP is used, the reserved virtual memory is released shortly after allocation (e.g., a short-lived per-request buffer), giving the attacker the opportunity to probe the target program multiple times. As detailed later, this allows an attacker to leak the size of the largest hole in the virtual memory address space and reduce the entropy of ASLR up to a *single* bit. The PAP, in turn, is based on reserving long-lived memory (e.g., a key-value store entry) and can be used in combination with the EAP to counteract the last bit of entropy or, by itself, to significantly reduce the entropy of ASLR.

dent. Any dependence naturally reduces the entropy.

4.1 Crafting primitives

The *ephemeral allocation primitive* (EAP) is available when a program allows *attacker-controlled input* to force the allocation of a short-lived memory object with an *attacker-controlled size*. In other words, the lifetime of the object must be such that the object is deallocated in a short amount of time (or by an attacker-controlled action, e.g. closing the connection that the object is associated with).

Even if an attacker induces the program to allocate a huge memory object of arbitrary size, such an allocation will succeed as large allocation operations typically result in an `mmap` system call. Thanks to demand paging, the system call returns right after reserving the required amount of virtual memory address space. The assignment of physical memory pages (page frames) to a virtual memory area, and even the population of the page tables, is only performed when a page is accessed for the first time. Hence, as long as the program does not immediately try to access all of the allocated virtual memory range, little physical memory is used and execution continues normally. This allows an attacker monitoring the program output to detect a *positive side effect* and verify that the corresponding address space was successfully reserved.

When the allocation size is larger than the amount of available virtual memory address space, however, such allocations will fail, typically causing the program to enter error-handling logic to return a particular error code (e.g., HTTP’s 500) to the client. This allows an attacker monitoring the program output to detect a *negative side effect* and verify that the allocation failed.

Other than monitoring program behavior for side effects, the attacker needs to fulfill two requirements to craft an EAP. First, the attacker needs to find an input which induces the program to allocate a short-lived memory object. This is, in practice, straightforward, since most programs allocate objects as part of their input-handling logic and release them afterwards. Second, the attacker needs to coerce the program to use an attacker-controlled size for the target object. While exploiting a naive program neglecting to set limits on the resources it will reserve (for instance, on the buffer size per client) is an option, in many cases the size of an allocation is calculated based on long-lived values which are stored in memory. As a result, an attacker in our threat model can rely on an arbitrary memory write vulnerability to corrupt one of those values and effect a memory allocation of a chosen size to craft an EAP. An example is an attacker able to corrupt a `buff_size` or similar global variable to control the size of a target allocation instance, a very common scenario in practice. In later sections, we substantiate this claim with empirical evidence on real-

world applications and present a methodology that can assist an attacker in the fast discovery of our primitives (Section 5).

To craft a *persistent allocation primitive* (PAP), an attacker can similarly abuse allocation instances and corrupt allocation sizes. The only difference is that a PAP relies on long-lived memory objects whose lifetime is not under attacker control. For example, a server program maintaining long-lived memory objects in a cache (spanning across several input requests) is amenable to a PAP, provided the attacker can control the allocation size. Oftentimes, however, the attacker can leverage the same primitive to obtain both an ephemeral and a persistent allocation primitive. For example, the common case of attacker-controlled allocations associated with individual client connections allow an attacker to craft either an EAP (when using nonpersistent connections) and a PAP (when using persistent connections) in a fully controlled way.

4.2 Breaking IH using the EAP

Many modern defenses depend on *information hiding* (IH) in order to protect a sensitive area which contains, for example, code pointers. We now discuss how a crafted EAP can reveal the *hidden* area (or hidden object), with few or even zero program crashes [10] and other detection-prone events [11] (hereafter, simply “crashes”). We discuss how an attacker can hide her traces even further (certainty of no crashes) in the next subsection. Here, we describe a simplified attack assuming that the defense randomizes the location of the hidden object within the largest available memory region. This assumption is fairly often verified in practice, given that if all holes in the address space are uniformly considered for hosting the hidden object, the largest hole (A) is, on average, 261 times more likely to be selected than the second largest hole. We later lift this and other assumptions on the address space organization in Section 4.3.

Once the hidden object is created, the hole size (A) is split into two new hole sizes, a large (L) and a small (S) one.² Assuming a random placement of a hidden object of size H in A , the bounds of L and S are $L_{min} = (A_{min} - H)/2$, $L_{max} = A_{max}$, $S_{min} = 0$, $S_{max} = (A_{max} - H)/2$. Hence, the distributions for the sizes L and S overlap. However, since in any given instance $L > S$ and, assuming the hidden object H is reasonably smaller than A , L is now the largest hole in the address space. Hence, an attacker can quickly identify L using EAP-based binary search (formalized in Algorithm 1, Appendix A). In detail, at each binary search iteration, the attacker performs a single EAP invocation for a given allocation size and

²For brevity, we omit explicit discussion of the case where $L = S$, which does not deviate from the common case where $L > S$.

observes its positive or negative side effects to select the allocation size for the next iteration. When the search completes, the attacker learns the largest allocation size and thus L . There can never be any confusion while we are performing the binary search for L as, if an allocation cannot be satisfied from the larger hole, it can certainly not be satisfied by the smaller one.

Since the hidden object is equally likely to have been placed below or above the midpoint of A , there’s is a 50% chance that L is the lower hole size. In this case, the location of the hidden object is precisely known: the base address of the hidden object is exactly located at `vm.mmap_min_addr+L`.

If L refers to the hole located higher than the hidden object, the uncertainty in the placement of the hidden object is the same as the uncertainty in the size of A . However, we can calculate the location of the hidden object based on the location of the `mmap` region. Given the interlinking of heap, stack, and code objects [18, 6, 1, 19], an attacker armed with an arbitrary memory read primitive can transitively explore allocated objects and discover the lowest memory mapped address. For example, in a typical quiescent application with a predictable memory layout, an attacker may simply leak `__libc_start_main`’s return address off the stack and immediately locate all the other virtual memory areas (VMAs) in the `mmap` region. Once the lowest memory mapped address `mmap_bottom` is known, the attacker can again deduce the location of the hidden object: its base address is exactly located at `mmap_bottom-L`.

Hence, the only uncertainty remaining is in the ordering of the two L - and S -sized holes, i.e., a *single* bit of entropy. In other words, an attacker probing the address space with an arbitrary memory read primitive has a 50% chance of discovering the hidden object on the first try and a 100% chance if she can tolerate a single invalid memory access. Even for nonforking server programs, if a process eventually gets restarted (either manually or automatically) with different randomization, the attacker has a 75% chance of guessing the hidden object’s location correctly after one crash, 87.5% with two crashes, 93.75% with three crashes and so on. While this attack is already fast and stealthy with great chances of going unnoticed in most practical settings, we show how to improve it even further without a single crash in the next subsection.

4.3 Using both the EAP and the PAP

When, in addition to the EAP, the attacker is in a position to employ a PAP as well (as it is often the case in practice), she can reliably break information hiding with no application crashes. In addition, she can locate the hidden object regardless of the original hole it was ran-

domly placed in (lifting our original assumptions). For simplicity, let us first consider the case of a hidden object placed somewhere in the middle of A , such that L and S are the first and second largest hole sizes in the address space. To recover L , the attacker begins by executing the EAP-only attack in Section 4.2. Subsequently, she can simply use the PAP to perform a L -sized allocation and eliminate the largest (L -sized) hole from the process' address space. Finally, she can repeat the same EAP+PAP strategy on the now largest allocation size in the address space to recover S and fill the remaining (S -sized) hole.

Since A is now completely hole-free, an attacker armed with an arbitrary memory primitive can reliably probe for the hidden object in the two possible locations in A and eliminate the remaining uncertainty. In detail, if the L -sized hole was at the beginning of the address space (and has now been filled by the PAP allocation), a read from `vm.mmap_min_addr+S` will be accessing zero-filled pages. If the S -sized hole was at the beginning of the address space, in turn, a read from `vm.mmap_min_addr+S` will be accessing pages containing data from the hidden object. In either case, by combining the EAP and the PAP, the attacker can easily disclose the location of the hidden object with no risk of crashes, quickly and stealthily exhausting information hiding's entropy.

Let us now reconsider our original assumption. In the general (if unlikely, for practical reasons³) case, the hidden object might be placed in a hole other than A . However, this is hardly a problem for an attacker armed with both the EAP and PAP. Such a zero-knowledge attacker can simply start with a single iteration of the EAP+PAP attack to fill the largest hole in the address space, then move to the second largest, and so on, until she can infer enough knowledge to first locate the hidden object's owning hole and then its location. For example, if the first largest possible hole identified is `sizeof(A)`, the attacker can learn the object is placed in either B or C . If the second largest possible hole identified is `sizeof(B)`, the attacker can learn the object is placed in C . At that point, she can perform the A -style EAP+PAP attack introduced earlier and locate the object with no crashes.

We note that exhausting the virtual address space with our iterative EAP+PAP attack strategy is not a concern in real-world scenarios. First, legitimate program allocations are normally satisfied by allocator arenas which rarely need to be extended during steady-state operations. In addition, the location of a target hidden object can in practice be determined without exhausting all the available holes. For example, an attacker could infer the location of a hidden object in A by only filling the L -sized hole and reliably reading from `vm.mmap_min_addr+L`.

³Placing the hidden object between stack and heap may impose unexpected limits on the growth of an application's data.

4.4 Using only the PAP

When allocations have to be persistent (e.g., only the PAP is available or the EAP has less desirable side effects), there are two main difficulties. First, given what we know about the hole size distributions, there might be multiple holes which can satisfy a request, but, without knowing their actual sizes, we cannot always tell which hole an allocation came from. Second, when an allocation succeeds, even if we know which hole it came from, we learn that that hole is at least as large. Contrary to the EAP though, we cannot retry a larger allocation size since we cannot "undo" the allocation.

An example serves to demonstrate. Suppose we start with the typical layout of a PIE executable (Table 1). Let us say we attempt an allocation of 130,500GiB and the allocation succeeds. The allocation was necessarily satisfied from hole A . We now have a lower bound on the size of A , yet we would like to find out its exact size. However, if we try to allocate a value in the range of 0-568GiB and the allocation succeeds, we cannot know whether the space was reserved in hole A or hole B as their size distributions now overlap.

We have designed and implemented a novel attack strategy which significantly reduces the uncertainty in the sizes of the holes. Our algorithm tracks the maximum allocatable size, as well as the allocated size, for each hole in what constitutes a *state*. Our approach then relies on two insights. First, it is highly preferential to probe using allocation sizes that can only be satisfied by a single hole. Second, when forced to perform an allocation which could have been satisfied from more than one hole, we need to fork and keep track of multiple states to model each feasible configuration of the holes in the targeted address space.

Building on these insights, our algorithm follows a cost-driven strategy to allow an attacker to select an optimal tradeoff between the number of allocation attempts and the entropy reduction obtained. We quantify this tradeoff in Section 7.5 and refer the interested reader to Appendix B for a detailed walkthrough of the formalized algorithm.

4.5 A more powerful EAP-only attack

Section 4.2 detailed how to locate the hidden object when it was placed in the largest address space hole (A). We then lifted this restriction by making use of the PAP in Section 4.3. An alternative way of stealthily probing for holes other than the largest one using the EAP only (when no PAP is available), is to try and trigger more than one EAP simultaneously. After having recovered L (Section 4.2), the attacker can simultaneously issue an allocation of L bytes while using a different allocation

request to probe for S . Even if the window is small, repeated simultaneous requests can make the chance of a false allocation arbitrarily small. When the program can afford more EAPs to be issued in parallel, the attack further improves, as the L -sized hole can be kept filled more reliably while a binary search is running to determine S . This approach generalizes to any number of holes.

4.6 Handling internal allocations

The attacks detailed in this section consider allocation primitives directly or indirectly based on `mmap`. However, when the primitive interacts with standard glibc allocation functions (e.g., `malloc`, `calloc`, `posix_memalign`, etc.), the result is one internal allocation for exceedingly large requests. Even though the requested size clearly does not fit in the largest available hole, glibc (version 2.19) allocates a new heap arena. The heap arena is allocated in the memory-mapped space and it is 64MiB-aligned. Therefore, the actual size of A that is recovered by a binary search differs from the previous end of the memory-mapped area by a random number which is 14 bits wide ($2^{26}/2^{12} = 2^{14}$).

Nevertheless, this is not a problem in practice, as the heap arenas form a circular, singly-linked, list. Therefore, an attacker armed with an arbitrary memory read primitive can navigate the links from the main allocation arena and discover all arenas in use (typically there would only be one link to follow). The main arena is a static variable in glibc, so it is located at a known (binary-dependent) offset from the highest address of the `mmap` space. Hence, as soon as the attacker leaks a pointer into the `mmap` space, she can easily account for the newly allocated heap arena as well.

5 Discovering Primitives

So far, in Section 4, we have discussed the mechanics of ephemeral and persistent memory-allocation primitives, which can assist an attacker in revealing the allocated ranges of a process in the virtual address space. In this section, we show that *dynamic data-flow tracking* (DFT) techniques applied to popular server programs can effectively assist attackers in discovering allocation instances that can potentially be abused to craft our primitives.

Discovering primitives that can result in powerful memory-allocation oracles involves identifying memory locations that, once controlled, can influence the input parameters of memory-allocation functions. Recall that, from Section 2, we assume attackers that are already in possession of (at least) one arbitrary read and write primitive. What the attacker lacks is a methodology to guide her to apply the read/write primitives and successfully craft EAPs and PAPs.

To model an attacker with arbitrary read/write control over memory, we start our analysis from a *quiescent state* of the program under attacker control. This state can be also manipulated and exercised over time by the attacker. As a simple example, consider a vulnerable web server. Assume the attacker can send a first a special-crafted HTTP request to trigger an arbitrary memory write vulnerability and gain control over memory. Next, the attacker issues a second request to invoke a memory allocation oracle. Therefore, in this particular example, our attacker-controlled quiescent state corresponds to that of an idle web server waiting for new requests. Once the second request is served, many parts of memory can be influenced either explicitly or implicitly. At one point, processing of the request triggers some memory-allocation function which serves as an oracle. It is important to stress that, depending on the server’s logic, parts of memory are overwritten (through successive allocations), while the request is processed. These parts cannot be generally controlled by the attacker using her arbitrary read/write primitives. However, the attacker *still* controls all the memory which was available in the original quiescent state (before the second request takes place). As long as memory of that state reaches a memory-allocation function, then the attacker can successfully use the oracle. Therefore, what we need to determine is the memory locations that influence memory-allocation sites and are *still* attacker-controlled, once the second HTTP request triggers memory-allocation functions.

Practically, this model can be easily realized using DFT. For our purposes, we use Memory-allocation Primitive Scanner (MAPScanner), a custom scanner based on libdft [20]. We start an application instrumented by MAPScanner, with all memory untainted, with *no* taint sources, and with all memory-allocation functions defined as sinks. Once the application is idle, we signal MAPScanner to taint all memory. At this point, the target quiescent state is defined and we assume that all memory is attacker-controlled. We then proceed and send a request to the server application. Any subsequent memory allocations that are triggered by the second request, since we have defined *zero* taint sources, wash out the taint of previous attacker-controlled memory. While the request is processed, MAPScanner reports all memory-allocation functions which are initiated with input from *still-tainted*, and therefore, *still-controlled* memory.

Notice, that, depending on the selected quiescent state and the input request, the attacker can discover more or fewer primitives. Using several complicated quiescent states, for example, those between handling two successive requests or between accepting the socket and receiving data, may uncover additional primitive candidates.

Once primitive instances are found, the attacker sim-

ply needs to locate the controlling data in memory (often a `buff_size` global variable originating from the configuration file), corrupt the data (and thus the allocation size) with an arbitrary memory write primitive, and monitor the execution for side effects. To classify a potential primitive as an EAP or PAP, the attacker will need to use the source or runtime experimentation to determine the lifetime of the corresponding allocated object along different program paths. Further, manual investigation is required to eliminate primitives that might not be usable because the value in memory is subject to additional validity checks in the attacker controlled paths.

In practice, we found that even when selecting the simplest quiescent state (i.e., idle server) and input (i.e., simple client request), an attacker can locate sufficient usable primitives to mount our end-to-end attacks (see Section 7).

6 Exploiting Timing Side Channels

Not all discovered primitives may automatically guarantee a realistic and crash-free attack. Certain types of primitives may not have any directly observable side effects (e.g., the server transparently recovering from allocation failures), making exploitation more complicated. Other types of primitives may result in program crashes (e.g., the server failing internal consistency checks), typically in both successful and unsuccessful cases, again making it difficult for an attacker to distinguish the two cases via direct observation. In both scenarios, however, an attacker can still infer the allocation behavior (success or failure) by measuring the time it takes to handle every particular request. We exemplify timing attack strategies for both the imperfect primitives presented above.

Even when a primitive has no directly observable side effects, allocation of memory and failure to allocate memory normally take a different amount of time. On Linux, for instance, a successful allocation is typically satisfied by a small VMA cache, avoiding lengthy walks of the red-black tree of virtual memory area (VMA) structures. However, on a VMA cache miss, before declaring an allocation failure, the kernel needs to walk all the nodes in the red-black tree in a compute-intensive loop, which takes measurably longer time to complete generating a *timing side channel* [21]. In fact, many kernel optimizations, such as VMA merging [22], explicitly seek to reduce the run-time impact of such expensive red-black tree walks. The timing signal becomes stronger for programs maintaining many VMAs and much stronger if the attacker can lure the program into allocating even more VMAs (however, VMA merging normally makes this difficult even for a PAP-enabled attacker). Even stronger timing side channels may be generated by the program itself. For example, to transparently recover

from allocation failures, the program may employ complex and time-consuming error-handling logic or log the event to persistent storage.

When a primitive results in program crashes in successful and unsuccessful cases, in turn, the presence and the strength of timing side channels is entirely subject to the internal cause of the crash. Interestingly, we found that the leading cause of crashes results in a very strong side channel. In fact, successful allocation-induced crashes are most commonly induced by a server attempting to fully initialize (or access) the huge allocated block, resulting in several time-consuming page faults before leading to the final out-of-memory error. As shown in Section 7, timing attacks which rely on crashes are remarkably effective in practice.

7 Evaluation

7.1 Primitive Discovery Results

We apply MAPScanner to a variety of well-known and popular server software. In particular, we consider BIND 9.9.3 (a DNS server), lighttpd 1.4.37 and nginx 1.6.2 (two popular web servers), as well as mysql 5.1.65 (a widely deployed database server). We built all programs using their default options (i.e. optimizations were enabled).

Since the presented applications have the form of a server accepting and servicing requests, we select, as the (simplest possible) attacker-controlled quiescent state, the point when the server is idle waiting for incoming connections, and, as the (simplest possible) attacker-controlled input, a default request to the server (Section 5). Of course, motivated attackers can carry out similar analyses starting from several additional quiescent states and inputs, so our results here are actually an (already sufficient) underapproximation of the real-world attack surface. Notice, finally, that we assume each server is being protected by an information-hiding-based defense mechanism which thwarts direct exploitation attempts (e.g., control-flow diversion).

Table 2 presents all the primitives discovered by our analysis. We name each instance of a primitive after the variable that an attacker needs to corrupt in memory to craft the corresponding allocation oracle. For each of the primitives, we report the type width of the memory-resident value that influences the allocation site. While 32-bit fields are only sufficient to bypass 32-bit (and not 64-bit) information hiding, we believe their availability can be indicative of the risks for 64-bit defense mechanisms—e.g., code refactoring changing an allocation size type to the common 64-bit `size_t` type may inadvertently introduce allocation oracles.

Table 2: For each particular application, we report the number of primitives found, the width of the allocation value, whether the primitive forces a crash, whether timing is necessary to determine success and if the primitive can be persistent as well. The “RE” (Residual Entropy) column assumes an attacker can reliably exploit the associated timing side channels. Values marked with (*) refer to lighttpd configured in forking mode.

	Primitive	Size	Crash-free	Timing-dependent	EAP	PAP	RE (bits)
bind	mgr->bpool	64-bit	✗	✓	✓	✗	1
	heap->size	32-bit	✓	✗	✓	Primarily	0
lighttpd	buffer->size#1	64-bit	✗	✓	✓	✗	0*
	buffer->size#2	64-bit	✗	✓	✓	✗	0*
	config_context->used	64-bit	✗	✓	✓	✗	0*
nginx	ls->pool_size	64-bit	✓	✗	✓	✓	0
	client_header_buffer_size	64-bit	✓	✗	✓	✗	1
	request_pool_size	64-bit	✓	✗	✓	✗	1
mysql	net->max_packet	32-bit	✓	✗	✓	✓	0
	net_buffer_length	32-bit	✓	✗	✓	✓	0
	connection_attrib	64-bit	✓	✗	✓	✓	0
	query_prealloc_size	64-bit	✓	✓	✓	✓	1
	records_in_block	32-bit	✓	✗	✓	✗	1

Additionally, we checked whether utilizing a primitive carried a risk of crashes (“crash-free” column). For primitives that did not provide directly observable side effects, the “timing-dependent” column indicates that the attacker needs to conduct a timing side channel attack to craft her primitives (we provide an example in Section 7.3). The EAP and PAP columns specify that the primitive can be used to perform an ephemeral and persistent allocation (respectively). Finally, we quantify the residual entropy after we perform the best attack at the attacker’s disposal for each primitive.

For each of these applications, our simple methodology was sufficient to discover 64-bit primitives able to quickly locate hidden objects with no residual entropy. In most cases, the discovered primitives were crash-free and could function as both EAPs and PAPs.

nginx and mysql are the best examples. They both provide ideal EAP+PAP attack primitives to stealthily bypass 64-bit information hiding with little effort. It is also worth noting that the `connection_attrib` primitive in mysql involves overwriting the requested stack size in a `pthread_attr_t` struct. As such, we expect a similar primitive to be available in all servers that create threads to service clients (either overwriting an application-specific attribute structure or the one in glibc).

For lighttpd, the server’s default configuration only allows the EAP-only attack, but, when the server is configured with forked worker processes, an attacker can successfully conduct the side-channel attack exemplified in Section 7.3 to eliminate all entropy and bypass information hiding.

Bind stands out as, depending on the server config-

uration, the `heap->size` primitive might be usable as an EAP or may effectively only function as a PAP. The reason for this behavior is that the effected allocation becomes part of a relatively long-lived cache. Hence, its lifetime is determined by administrator choices and performance considerations. When cached objects are not eagerly expired, the primitive may only be usable as a PAP for the duration of a practical attack.

Overall, our simple analysis shows that real-world information-hiding-protected applications stand very little chance against attackers armed with allocation oracles.

7.2 EAP+PAP attack on nginx

To illustrate how the combined EAP+PAP attack works in practice, we consider the `ls->pool_size` primitive discovered during our investigation of the nginx web server (Table 2).

When servicing a new connection, nginx’s `ngx_event_accept()` function allocates a per-connection memory pool (`c->pool`) using the size stored in the listening socket data structure associated to the socket the `accept()` originated from. `ngx_event_accept()` instantiates the pool by calling out to `ngx_create_pool()`, which eventually allocates the required memory by means of `posix_memalign()`.

Using our primitive discovery methodology, we were easily able to determine that the `size` argument to `posix_memalign()` originated from a value resident in live memory for our idle attacker-controlled quiescent state. This means that an arbitrary memory write vulnerability in any of the code that processes untrusted in-

put can be used to overwrite this value with an attacker-selected size, once the memory location is known.

We then verified that `ls->pool_size` is trivially accessible by following the `ls` field of the `ngx_connection_t` structure, a pointer to which is always available on various stack locations while the server is executing request-processing code.

Using this information, the attacker is able to craft an ephemeral allocation primitive by using an arbitrary read to navigate the pointer chain until she determines the address of `ls->pool_size`. At this point, she can effect a call to `posix_memalign()` with a size of her choosing by overwriting `ls->pool_size` and then opening a connection to the server. If the allocation request was successful, the attacker can issue an HTTP request over that new connection (positive side effect). If the allocation cannot be accommodated, the connection is forcibly closed by the server (negative side effect).

Using the same procedure, the attacker can craft a PAP by simply keeping the connection open in the last step. To conduct the complete attack, the attacker first employs the EAP to determine the size of the larger of the two holes around the hidden object (for simplicity, we only discuss the case when the hidden object is placed in the largest contiguous pre-existing hole; other scenarios are investigated in Section 4.3). Having determined the maximum allocation (i.e., hole) size, she relies on the PAP to allocate the exact size of the larger hole, taking it out of the picture. She then proceeds to conduct the EAP-based attack against the smaller hole around S . Finally, she simply probes at address `vm.mmap_min_addr+S` to complete the attack, as described in Section 4.3.

7.3 Timing-based attack on lighttpd

Next, we focus on the execution of an EAP-only attack which relies on a timing side channel. To demonstrate such an attack, we rely on the `config_context->used` primitive in lighttpd. In order craft this primitive, we configured lighttpd to use worker processes by setting the `server.max-worker` configuration variable to a non-zero value. With no loss of generality, we limit our analysis to one worker process, as an arbitrary memory access primitive makes it a matter of book-keeping to tag the workers (e.g. by writing a different value for each worker to an unused memory location), so that the attack code can target a single process.

Again using our primitive discovery methodology, we easily determined that `srv->config_context->used` is used as an argument to `calloc()` in the body of `connection_init()`. Similarly, we showed that pointers to `srv` are available in the stack frames above the event loop, which renders `srv->config_context->used` trivially accessi-

ble to an attacker equipped with arbitrary memory read/write primitives.

The second argument to `calloc()` at this call site is `sizeof(cond_cache_t)`, which amounts to 144 bytes. Since that is less than the default page size on x86, we can always find a value that will result in the allocation of any given number of pages.

Crucially, the return value from `calloc()` is never checked for failure. Therefore, the only way to determine whether the allocation succeeded or not is to send a simple request so as to drive the server to a path which will dereference the pointer. That path is simply `http_response_prepare() -> config_cond_cache_reset()`, which will iterate over all elements of the array. As a result, if the allocation fails, the worker process *immediately* crashes on trying to access the first element, resulting in a closed connection for the client. If the allocation succeeds but the allocated size is much larger than the amount of physical memory on the system, this allocation incurs *several lengthy page faults* before causing an out-of-memory (OOM) condition—on which Linux’ “OOM killer” terminates, with high reliability, the worker process. If the system can survive faulting in all the allocated pages (presumably because the allocation was “small”), the server eventually sends back a response. Using either the timing or the reception of an HTTP response, we can infer whether the attempted allocation succeeded or failed.

When the worker process crashes or is terminated by the OOM killer, the parent is notified and forks a new child in replacement (indeed one of the motivations for using worker processes in server software is for crash recovery purposes). Each new worker process inherits the address space of the parent; hence, all memory regions (including the hidden object) remain at stable virtual addresses across worker restart events.

Given the several page faults incurred before a crash, the timing side channel we rely on yields a very strong signal. In our testing, we performed the attack 40 times and were able to reliably differentiate between a successful and a failed allocation in all of them. By using the same primitive as a PAP (as is possible in this configuration), we were able to persistently allocate the recovered size for the largest hole and then repeat the EAP attack on the smaller hole as done earlier. In summary, by relying on EAP+PAP primitives and a strong timing side channel we could successfully recover the address of the hidden object and bypass information hiding in all cases in our experiments.

7.4 EAP-only attack

We evaluated the accuracy and performance of the EAP in defeating the information-hiding properties of ASLR

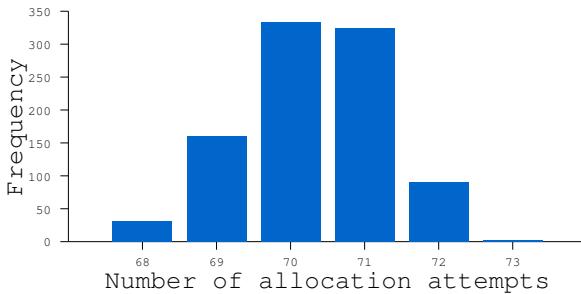


Figure 2: Histogram of the number of operations for recovering the exact hole sizes around a hidden object in an nginx executable [n=1000]

by preallocating a hidden object of a size of 2MiB in the address space of nginx (compiled as a position-independent executable) and then trying to determine the sizes of the larger and smaller holes on either side of it, as described in Section 7.2. Taking into account the complications and workarounds described in Section 4.6, we were able to exactly determine the size of both the larger and smaller hole and subsequently uncover the exact location of the hidden object, without incurring any invalid memory accesses.

Figure 2 depicts the number of required allocation attempts over 1000 runs (using different random configurations). On localhost and using gdb to effect the arbitrary memory access, the attack completed after an average of 28.20s with a median of 28.21s.

7.5 PAP-only attack

When the only primitive available to the attacker is the PAP, she needs to consider a number of tradeoffs. Clearly, the attacker is interested in reducing her entropy with respect to the position of the hidden region in the targeted address space. At the same time, different considerations might cause her to strive for minimal or rapid interaction with the target process. For example, a very large number of requests to a remote server might very well increase the chance that the attack will be noticed by network intrusion detection systems. Similarly, as the duration of the attack increases, so does the chance that unrelated process activities, such as servicing requests for other clients or periodically scheduled work, may interfere with the workings of the algorithm.

There exist two tunable parameters that affect the behavior of our PAP-only attack. One selects between the number of allocation attempts and the entropy reduction obtained, the other between entropy reduction and risk of failure.

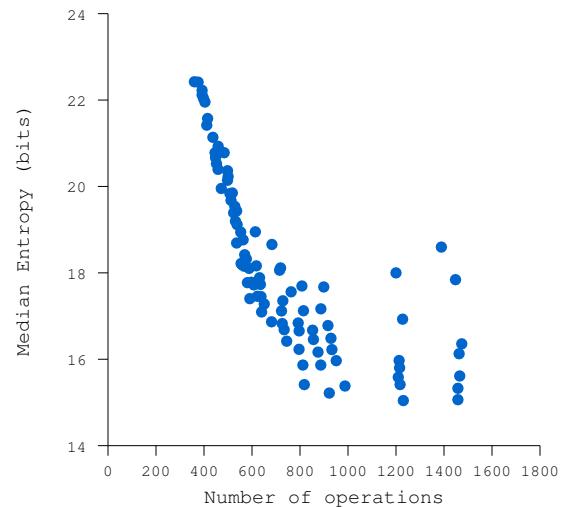


Figure 3: Parameter space exploration for the PAP-only attack

Figure 3 presents the number of allocation attempts versus the *median* of the residual entropy in the size of the largest hole after the completion of the attack, for every pair of parameter values that we explored. We can observe that extreme choices of the parameter values result in pathological behavior, either in the number of operations or in the residual entropy or both.

Conversely, there exist better parameter choices which do reasonably well for both metrics. Appendix C goes into more depth on the parameter values and their effect on the behavior of the attack.

Observe that, in attack scenarios where a number of requests on the order of a thousand is acceptable, there exist several parameters for which the median residual entropy is reduced to 15 bits. Notice also that when the size of the hidden region can be expected to be comparable to the residual entropy, the chances of successfully probing for the region are significantly increased.

For a round-trip time of 1s, even the PAP-only attack would take at most half an hour (12 minutes on average), which is still an eminently practical amount of time in many real-world settings.

7.6 Effectiveness against modern defenses

The presented attacks change the picture for the protection offered by state-of-the-art defense mechanisms that rely on information hiding. CPI’s safe area [3] and many other prior solutions [4, 23, 19, 24] rely on information hiding to protect a single hidden region. For all such solutions, our attacks in Section 4 apply directly and can

locate the hidden region with little or no crashes. Prior work has demonstrated a memory probing-based crash-free CPI bypass with roughly 110,000,000 operations on nginx [12]. Using our combined EAP and PAP crash-free attack, we can replicate their attack needing only 74 operations in the worst case (Fig. 2). This is a 1,400,000x improvement in attack efficiency, which, projected on the request time reported in [12], translates to 0.23s (rather than 97 hours) to locate the hidden region. In addition, our crash-free attack is even faster than the fast crash-prone attack presented in [12] (6s with 13 crashes).

More recent client-side probing attacks [11] offer similar guarantees (i.e., locating CPI’s safe area in 32 probes), but their probing strategy relies on exception handling rather than crash recovery, ultimately improving the attack efficiency. We note that both existing probing attacks [12, 11] exploit assumptions on CPI’s huge hidden region size (on the order of 2^{42} bytes when using a sparse table and $2^{30.4}$ when utilizing a hash table [11]) to reduce the entropy and make the attack practical. In stark contrast, our attacks make no assumptions on the region size, and doing so would allow even a PAP-only attack to succeed without crashes.

Other solutions, such as ASLR-Guard [6], SafeStack [3], and other shadow stack implementations [25], rely on information hiding to protect multiple hidden regions. For example, all the shadow stack solutions need to maintain a per-thread hidden region. We note that our attacks generalize to multi-region information hiding with essentially the same impact. In particular, while multi-threaded programs disqualify the simple EAP-only attack, our best (EAP+PAP) attack naturally extends to multi-region solutions and can quickly bypass them (although more allocations may be required).

Finally, many leakage-resilient defenses [26, 27, 28, 29] enforce execute-only memory to protect the hidden (code) region from read-based disclosure attacks [1]. However, such defenses are susceptible to execution-based disclosure attacks in crash-tolerant applications [10]. To counter such attacks, some solutions deploy booby traps in out-of-band trampolines [26, 27]. With allocation oracles, an attacker can sidestep the booby-trapped trampolines and quickly find the hidden region, enabling more practical and guided execution-based disclosure attacks against such defenses.

8 Mitigations

One strategy to defend against allocation oracles is to enforce an upper limit on the maximum amount of virtual memory that a process can allocate. This mechanism is already available on Linux (and other POSIX-compatible operating systems) via the `RLIMIT_AS` resource limit (adjustable via `setrlimit`). Setting this limit to a small,

though still sufficient for most current applications, value would thwart any attempts to probe the sizes of the larger holes in the address space. The resource limit can be set by the application itself (in which case, a defense mechanism could intercept it and adjust it to accommodate its own needs for virtual addresses) or it can be hard-capped by the administrators without any need for program adjustments. The main difficulty lies in predicting the maximum virtual address space usage under all conceivable conditions so as to never deny legitimate allocation requests for production applications. Nonetheless, its wide availability, straightforward deployment, and robustness (see below) make `RLIMIT_AS` our primary recommendation for compatible workloads and configurations.

For some classes of applications (especially those relying on memory overcommit), limiting the amount of virtual address space available may be problematic, e.g., when memory-mapping huge files. In such cases, one could switch to a strict overcommit policy and have the applications always use `mmap`’s `MAP_NORESERVE` flag for huge—but known to be benign—allocations. `MAP_NORESERVE` instructs the kernel not to count the corresponding allocations towards the overcommit limit. However, this mitigation strategy would still allow an attacker to inject the `MAP_NORESERVE` flag in `mmap` calls using memory-resident arguments and craft our primitives. Another issue with such a strict overcommit strategy is that it is incompatible with memory-hungry applications that rely on `fork` and cannot simply switch to `vfork` (the `redis` server being a prime example). This problem can only be directly mitigated with the addition of a new flag to the `clone` system call to mimic the semantics of `MAP_NORESERVE`.

In some setups, one may deploy an IDS looking for anomalous events (i.e., allocations) in a given application [30]. However, this approach generally requires per-application policies (e.g., only allow huge allocations of a specific size). A policy looking for frequent huge allocations in arbitrary applications is more generic but problematic, as an attacker can easily dilute the very few probing attempts required by our attacks over time [30].

Finally, defense mechanisms could bracket their hidden regions with randomly-sized trip hazard areas [31] to deter in-region memory probing. This is the immediate systemwide mitigation we recommend for information-hiding-based solutions already deployed in production [32]. Albeit still probabilistic (and thus prone to attacks), such solution can also provide efficient protection against other (known) side-channel attacks [31].

9 Related work

We distinguish between approaches that aim at breaking ASLR in general and approaches that try to

break more advanced defense techniques that rely on ASLR-based information hiding.

Breaking ASLR has been fertile research ground for years and became especially popular in recent years. From the outset [30], pioneering work showed that the randomization in 32-bit address spaces provide insufficient entropy against practical brute-force attacks, so we focus on 64-bit architectures (x86-64) in this section.

In practice, bypassing standard (i.e., coarse-grained, user-level) ASLR implementations is now common. For an attacker, it is, for instance, sufficient to disclose a single code pointer to de-randomize the address space [33]. Even fine-grained ASLR implementations [34] are vulnerable to attacks that start with a memory disclosure and then assemble payloads in a just-in-time fashion [1].

More advanced attack vectors rely on side channels via shared caches. Specifically, recently accessed memory locations remain in the last-level cache (LLC) which is shared by different cores on modern x86-64 processors. As it is much faster to access memory locations from the cache rather than from memory, it is possible to use this timing difference to create a side channel and disclose sensitive information. By performing three types of PRIME+PROBE attacks on the CPU caches and the TLB, Hund et al. [35] could completely break kernel-level ASLR by mapping the entire virtual address-space of a running Windows kernel. To perform a PRIME+PROBE attack, the attacker needs the mapping of memory locations to cache sets. In modern Intel processors, this mapping is complex and reverse engineering requires substantial effort [35]. However, performance counter-based and other techniques have been proposed to lower the reverse engineering effort [36].

Even without *a priori* disclosures, attackers may still break ASLR using Blind ROP (BROP) [10]. A BROP attack sends data that causes a control transfer to another address and observes the behavior of the program. By carefully monitoring server program crashes, hangs, or regular output, the attacker can infer what code executed and, eventually, identify ROP gadgets. After many probes (and crashes), she gets enough gadgets for a ROP chain. BROP is a remote attack method applicable (only) to servers that automatically respawn upon a crash.

In general, leaking information by means of side channels is often possible. To launch such an attack, an attacker typically uses memory corruption to put a program in a state that allows her to infer memory contents via timings [21, 37, 12] or other side channels [10].

As ASLR by itself does not provide sufficient protection against the attacks described above, the community is shifting to more advanced defenses that build on ASLR to hide sensitive data (such as code pointers) in a hidden region in a large address space, typically not referenced by any pointers within the attacker’s reach.

Hiding secret information in a large address space is now common practice in a score of new defenses. For example, Oxymoron [4] protects the *Randomization-agnostic Translation Table* (RaTTle) by means of information hiding, and Opaque CFI [23] protects the so-called *Bounds Lookup Table* (BLT) in a similar way. Likewise, Isomeron [19] keeps the *execution diversifier data* secret and StackArmor [25] isolates potentially vulnerable stack frames by means of hiding in a large address space. Finally, on x86-64 architectures, CFCI [24] also needs to hide a few MBs of protected memory.

One of the best-known examples of a defense that builds on ASLR-backed information hiding is Code Pointer Integrity [3]. CPI splits the address space in a standard and a safe region and stores all code pointers in the latter, while restricting accesses to the (huge) safe region to CPI-intrinsic instructions. Moreover, it also provides every thread with a shadow stack (called SafeStack in CPI) in addition to the regular stack and uses the former to store return addresses and other proven-safe objects. Both the shadow stacks (which are relatively small) and the safe region (which is huge) are hidden at a random location in the virtual address space.

By means of probing on a timing side channel, Evans et al. showed that it is possible to circumvent CPI and find the safe region [12]. However, depending on the construction of the safe region, the attack may require a few crashes or complete in several hours in order to be stealthy (i.e., crash-free). Moreover, similar to the recent CROP [11] (which instead relies on specially crafted crash-resistant primitives), this attack needs to resort to full memory probing to locate small hidden regions (unlike CPI’s) in absence of implementation flaws. Full memory probing forces the attacker to trigger many crashes and other detection-prone events, and its efficiency quickly degrades when increasing the address space entropy.

Concurrent work [31] relies on *thread spraying* to reduce the entropy in finding a per-thread hidden object. Allocation oracles can make thread spraying attacks faster by providing a more efficient disclosure primitive compared to the memory probing primitives used in [31].

Unlike all the existing attacks, *allocation oracles* demonstrate that an attacker can craft pervasively available primitives and locate the smallest hidden regions in the largest address spaces, while leaving little or no detectable traces behind.

10 Conclusions

We have shown that information hiding techniques that rely on randomization to bury a small region of sensitive information in a huge address space are not safe on modern Linux systems. Specifically, we introduced new in-

formation disclosure primitives, allocation oracles, that allow attackers to probe the holes in the address space: by repeated allocations of large chunks of memory, the attacker discovers the sizes of the largest areas of unallocated memory. Knowing the sizes of the largest holes greatly reduces the entropy of randomization-based information hiding and allows an attacker to infer the location of the hidden region with few to no crashes or noticeable side-effects. We have also shown that allocation oracles are pervasive in real-world software.

Unfortunately, information hiding underpins many of the most advanced defense mechanisms today. Without proper mitigation, they are all vulnerable to our attacks. While one may deploy more conservative memory management policies to limit the damage, we emphasize that the problem is fundamental in the sense that allocation oracles always reduce the randomization entropy, regardless of the mitigation and the address space size. In general, information hiding is vulnerable to entropy reduction by whatever means and it is not unlikely that attackers can combine allocation oracles with other techniques. In our view, it is time to reconsider our dependency on the pseudo-isolation offered by randomization and opt instead for stronger isolation solutions like software fault isolation or hardware protection.

11 Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to the relevant parties.

12 Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was supported by the European Commission through project H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 644571 and by Netherlands Organisation for Scientific Research through project NWO 639.023.309 VICI “Dowsing”.

References

- [1] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monroe, and A.-R. Sadeghi, “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization,” in *IEEE S&P ’13*.
- [2] T. H. Dang, P. Maniatis, and D. Wagner, “The performance cost of shadow stacks and stack canaries,” in *ASIACCS ’15*.
- [3] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *OSDI’ 14*.
- [4] M. Backes and S. Nürnberg, “Oxymoron: Making fine-grained memory randomization practical by allowing code sharing,” in *USENIX Security ’14*.
- [5] PaX Team, “Address space layout randomization (ASLR),” 2003, <http://pax.grsecurity.net/docs/aslr.txt>.
- [6] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, “ASLR-Guard: Stopping address space leakage for code reuse attacks,” in *CCS ’15*.
- [7] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *SOSP ’93*.
- [8] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc, “Native client: A sandbox for portable, untrusted x86 native code,” in *IEEE S&P ’07*.
- [9] L. Deng, Q. Zeng, and Y. Liu, “ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries,” in *IFIP SEC ’15*, 2015.
- [10] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, “Hacking blind,” in *IEEE S&P ’14*.
- [11] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, “Enabling client-side crash-resistance to overcome diversification and information hiding,” in *NDSS ’16*.
- [12] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiropoulos-Douskos, M. Rinard, and H. Okhravi, “Missing the point(er): On the effectiveness of code pointer integrity.”
- [13] “CVE-2015-3864,” <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3864>.
- [14] D. Magenheimer, “Memory overcommit... without the commitment,” in *Xen Summit*, 2008.
- [15] “Redis administration,” <https://web.archive.org/web/20150905213905/http://redis.io/topics/admin>.
- [16] E. Sammer, *Hadoop Operations*, 2012, ch. 4.
- [17] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, “kGuard: Lightweight kernel protection against return-to-user attacks,” in *USENIX Security ’12*.
- [18] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, “Losing control: On the effectiveness of control-flow integrity under stack attacks,” in *CCS ’15*.
- [19] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monroe, “Isomeron: Code randomization resilient to (just-in-time) return-oriented programming,” in *NDSS ’15*.
- [20] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, “Libdfit: Practical dynamic data flow tracking for commodity systems,” in *VEE ’12*.
- [21] J. Seibert, H. Okhravi, and E. Söderström, “Information leaks without memory disclosures: Remote side channel attacks on diversified code,” in *CCS ’14*.
- [22] “Mmap speedup,” http://www.verycomputer.com/180_d89089d5a857ed08_1.htm.
- [23] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, “Opaque control-flow integrity,” in *NDSS ’15*.
- [24] M. Zhang and R. Sekar, “Control-flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks,” in *ACSAC ’15*.
- [25] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, “StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries,” in *NDSS ’15*.
- [26] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz, “Readactor: Practical code randomization resilient to memory disclosure,” in *IEEE S&P ’15*.

- [27] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, “It’s a TRaP: Table randomization and protection against function-reuse attacks,” in *CCS ’15*.
- [28] J. Gionta, W. Enck, and P. Ning, “HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities,” in *CODASPY ’15*.
- [29] C. L. Kjell Braden, Lucas Davi and M. F. P. L. Ahmad-Reza Sadeghi, Stephen Crane, “Leakage-resilient layout randomization for mobile devices,” in *NDSS ’16*.
- [30] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *CCS ’04*.
- [31] E. Göktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, “Undermining information hiding (and what to do about it),” in *USENIX Security ’16*.
- [32] “SafeStack,” <http://clang.llvm.org/docs/SafeStack.html>.
- [33] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, “Breaking the memory secrecy assumption,” in *EuroSec ’09*.
- [34] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Enhanced operating system security through efficient and fine-grained address space randomization,” in *USENIX Sec ’12*.
- [35] R. Hund, C. Willems, and T. Holz, “Practical timing side channel attacks against kernel space ASLR,” in *IEEE S&P ’13*.
- [36] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon, “Reverse engineering Intel last-level cache complex addressing using performance counters,” in *RAID ’15*.
- [37] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, “Dedup est machina: Memory deduplication as an advanced exploitation vector,” in *IEEE S&P ’16*.

A Base EAP-only algorithm

Algorithm 1 Binary search using the Ephemeral Allocation Primitive; sizes are in pages.

```

function DEDUCE(low, high)
  if low = high then
    return low
  if high - low = 1 then
    res  $\leftarrow$  TEMP-ALLOC(high)
    if SUCCESS(res) then
      return high
    else
      return low
    midpoint  $\leftarrow$   $\lfloor (\text{high} + \text{low}) / 2 \rfloor$ 
    res  $\leftarrow$  TEMP-ALLOC(midpoint)
    if SUCCESS(res) then
      return DEDUCE(midpoint, high)
    else
      return DEDUCE(low, midpoint - 1)
  
```

Hole	Total	Max
A	0B	131068GiB
B	0B	1028GiB
C	0B	4GiB

Table 3: Initial state for a PIE executable

B PAP-only algorithm

For each hole G , we maintain two variables, G_{total} and G_{max} . The first variable tracks the number of bytes allocated from G . The second holds the maximum number of bytes that may still be allocatable from G at any point in time. So when an allocation of size S is known to originate from G , we increase G_{total} by S and decrease G_{max} by S . Crucially, when an allocation of size S fails, we know that no hole has S bytes available. Therefore, the *max* variable for every tracked hole needs to be adjusted to S minus the pagesize (see the functions HOLE-SATISFIED, HOLE-FAILED-TO-SATISFY, called for holes that satisfied or failed to satisfy an allocation request, respectively).

```

function HOLE-SATISFIED(size, G)
   $G_{max} \leftarrow G_{max} - size$ 
   $G_{total} \leftarrow G_{total} + size$ 
function HOLE-FAILED-TO-SATISFY(size, G)
  if  $G_{max} > size - \text{pagesize}$  then
     $G_{max} \leftarrow size - \text{pagesize}$ 
  
```

A state consists of the set of *max* and *total* variables for each tracked hole. In the initial state we may have some information for the maximum size of each hole, but no bytes have been allocated during the run of our algorithm, so that $G_{total} = 0, \forall G$. Given the size distributions in Table 1, the initial state for a simple PIE executable is as given on Table 3.

Descent mode The algorithm operates in two modes. Suppose that the highest maximum value (*hmv*) is unique across all the tracked holes and G is the hole it is associated with (i.e. $\#H : G_{max} = H_{max}$). In this case, we try decreasing allocation sizes which can only be satisfied by G , in the hope that an allocation will succeed, causing G_{max} to be decremented below the next-highest maximum (*nhmv*) value so that we will remain in this mode for the next step of the algorithm.

There is an inherent tradeoff between the accuracy and the number of allocations we try. In the extreme, we could explore the interval $[nhmv, hmv]$ by starting with *hmv* and decreasing the allocation size by one page after each failed attempt. Of course, this would result in a huge number of allocation attempts, rendering the approach impractical.

```

function CALCULATE-STEPS(high, low)
    size  $\leftarrow$  high  $-$  low
    if size = pagesize then
        return [high]
    step  $\leftarrow$  size/split
    sizes  $\leftarrow$  []
    idx  $\leftarrow$  0
    for n in 0..(split  $-$  1) do
        sz  $\leftarrow$  high  $-$  n * step
        rem  $\leftarrow$  sz % pagesize
        if rem > 0 then
            sz  $\leftarrow$  sz  $-$  rem + pagesize
        if idx = 0  $\vee$  sizes[idx  $-$  1]  $\neq$  sz then
            sizes[idx]  $\leftarrow$  sz
            idx  $\leftarrow$  idx + 1

```

While larger successful allocations are desirable, we elect to trade some resolution for a reduction in the number of attempts necessary. The way we do this is by selecting a *split* factor for the interval $[nhmv, hmv]$, and trying decreasing allocations with an (approximate) step of $\frac{hmv - nhmv}{split}$ bytes (special considerations need to be made for respecting page boundaries; see the CALCULATE-STEPS function). A larger split factor results in more allocation attempts but higher chances of quickly minimizing the *max* variable of G.

At every given step, the allocation might succeed (in which case we update G_{total} and G_{max}) or it might fail and we appropriately reduce every *max* variable to just below the failed allocation size. This means that the difference between the current *hmv* and the *nhmv* keeps shrinking as allocations fail.

For reasons that will become apparent soon, we are willing to expend more allocation attempts to avoid the situation when *hmv* becomes equal to the *nhmv*. Therefore, when the last step above the *nhmv* results in a failed allocation, we reiterate the algorithm, again splitting the interval between the current *hmv* and the *nhmv* according to the split factor and trying descending allocation size using a new, smaller, step size. The algorithm continues trying ever smaller allocations using an ever finer step size, until the allocation of *nhmv* + pagesize bytes. If that allocation fails, then we have to switch into the mode where there are multiple highest maximum values (Algorithm 2, line 34).

Forking mode We are now in a state where there exist *n* holes, $G^1, \dots, G^n : G_{max}^1 = \dots = G_{max}^n = hmv$, i.e. the *hmv* has multiplicity *n*. The only way to make progress is to try an allocation smaller than *hmv*; yet if the allocation succeeds, we are not in a position to tell which hole the bytes were allocated from. What's worse, more than one hole might be able to accommodate the allocation we attempt.

Algorithm 2 Decision

```

1: function STATE-MAXES(s)
2:   res  $\leftarrow$   $\emptyset$ 
3:   for all max  $\in$  MAXES(s) do
4:     res  $\leftarrow$  res  $\cup$  max
5:   sorted_maxes  $\leftarrow$  SORT-DESCENDING(maxes)
6:   groups  $\leftarrow$  GROUP-BY-MAX(sorted_maxes)
7:   result  $\leftarrow$   $\emptyset$ 
8:   for all g  $\in$  groups do
9:     result  $\leftarrow$  result  $\cup$  (ANY-MAX(g), COUNT(g))
10:  return result
11: function DETERMINE-GROUPS(states)
12:   maxes  $\leftarrow$   $\emptyset$ 
13:   for all s  $\in$  states do
14:     maxes  $\leftarrow$  maxes  $\cup$  STATE-MAXES(s)
15:   maxes  $\leftarrow$  SORT-DESCENDING(maxes)
16:   groups  $\leftarrow$  GROUP-BY-MAX(maxes)
17:   result  $\leftarrow$   $\emptyset$ 
18:   for all g  $\in$  groups do
19:     g  $\leftarrow$  SORT-DESCENDING-BY-MULTIPLICITY(g)
20:     maxval  $\leftarrow$  FIRST(g)
21:     result  $\leftarrow$  result  $\cup$  maxval
22:   return result
23: function DECIDE(states)
24:   maxvals  $\leftarrow$  DETERMINE-GROUPS(states)
25:   (hmv, m)  $\leftarrow$  FIRST(maxvals)
26:   if hmv  $\leq$  mshs then
27:     return states
28:   if COUNT(maxvals) = 1 then
29:     nhmv  $\leftarrow$  mshs + pagesize
30:     if hmv = nhmv then
31:       return states
32:     sizes  $\leftarrow$  CALCULATE-STEPS(hmv, nhmv)
33:     states  $\leftarrow$  DESCEND(states, m, sizes)
34:     return states
35:     (nhmv,)  $\leftarrow$  SECOND(states)
36:     if nhmv  $\leq$  mshs then
37:       nhmv  $\leftarrow$  mshs + pagesize
38:     if hmv = nhmv then
39:       return states
40:     sizes  $\leftarrow$  CALCULATE-STEPS(hmv, nhmv)
41:   return DESCEND(states, m, sizes)

```

This constitutes a second mode of operation for our algorithm. After we select an allocation size T , we attempt to allocate T bytes n times. If all n allocations succeed, the max values for the n holes all get reduced by the same amount; if the new maximum values are still higher than any other maximum value, we remain in the same mode. If there is now a unique (necessarily different) hm , we switch modes.

We then consider the case when k out of n allocation attempts succeed. The allocations could have come out of any set of k holes and there are $\binom{n}{k}$ possible ways to pick the successful holes out of the n we started with. At this point, we fork the current state into $\binom{n}{k}$ new ones, one for each possible combination. In each newly-created state, k holes get their *total* variables incremented and their *max* variables decremented by the allocated size, whereas the maximum values of the remaining $n - k$ states are lowered to below the allocated size (as we consider the allocation from those holes to have been a failure). After a fork, we are left with a number of active states, one of which matches the actual system state as regards the total allocated bytes for each hole.

Finally, when all n allocations fail, we need to pick a new, smaller allocation size. The obvious approach would be to halve the size for the next allocation attempt. However, this choice leads to a pathological situation. Recall that every failed allocation attempt causes all maximum variables to be set to one page below the attempted value so that, as long as the allocations all fail, the maximum values are reduced in lockstep. Consider the case when an allocation succeeds; the new maximum value for a hole G which is considered successful is set to $G'_{max} = G_{max} - allocation_size = G_{max} - G_{max}/2 = G_{max}/2$. Regarding a hole H for which the allocation was considered a failure, the *max* variable is updated to $H'_{max} = allocation_size - pagesize = H_{max}/2 - pagesize$. Since G_{max} was equal to H_{max} , the new maximum values for all the n holes will all be within a page of each other, which makes it all but certain that after one descent step we will re-enter the forking mode, which increases the chances of a combinatorial explosion in the number of active states.

To increase our chances switching back to descent mode, we elect to pick the next allocation size exactly as we do when the hm is unique.

Generalized Algorithm We extend the algorithm described above to operate when there is more than one active state. Our driving concern is to be able to differentiate between active states. To that end, we collect the highest maximum value of each state. We then attempt a descent from the highest maximum value to the next-highest maximum value using the split factor, as described for the single-state case.

Algorithm 3 Descent

```

function ALLOCATE( $m$ )
   $count \leftarrow 0$ 
  for  $i$  in  $0..m$  do
    if ALLOC() then
       $count \leftarrow count + 1$ 
  return  $count$ 

function NSTATE( $size, rest, satisfied, not\_satisfied$ )
   $holes \leftarrow rest$ 
  for all  $G$  in  $satisfied$  do
     $holes \leftarrow holes \cup$  HOLE-SATISFIED( $size, G$ )
  for all  $G$  in  $not\_satisfied$  do
     $holes \leftarrow holes \cup$  HOLE-NOT-SATISFIED( $size, G$ )
  return CREATE-STATE( $holes$ )

function DO-COMBINE( $rest, selected, previous, count, candidates, accum$ )
  if  $count = 0$  then
     $nstate \leftarrow$  NSTATE( $size, rest, selected, previous$ )
     $accum \leftarrow accum \cup nstate$  return  $accum$ 
  else if EMPTY( $candidates$ ) then return  $accum$ 
  else
     $x \leftarrow$  FIRST( $candidates$ )
     $candidates \leftarrow TAIL( $candidates$ )$ 
     $accum \leftarrow$  DO-COMBINE( $rest, selected \cup x, previous, count - 1, candidates, accum$ ) re-
  turn DO-COMBINE( $rest, selected, previous \cup x, count, candidates, accum$ )

function COMBINE( $rest, count, candidates$ )
  DO-COMBINE( $rest, \emptyset, \emptyset, count, candidates, \emptyset$ )

function UPDATE-STATES( $states, size, count, m$ )
   $nstates \leftarrow \emptyset$ 
  for all  $s \in states$  do
     $candidates \leftarrow \emptyset$ 
     $rest \leftarrow \emptyset$ 
    for all  $G \in$  HOLES( $s$ ) do
      if  $G_{max} \geq size$  then
         $candidates \leftarrow candidates \cup G$ 
      else
         $rest \leftarrow rest \cup G$ 
    if COUNT( $candidates$ )  $\geq count$  then
       $res \leftarrow$  COMBINE( $rest, count, candidates$ )
       $nstates \leftarrow nstates \cup res$ 
  return  $nstates$ 

function DESCEND( $states, m, sizes$ )
  for  $size$  in  $sizes$  do
     $count \leftarrow$  ALLOCATE()
     $states \leftarrow$  UPDATE-STATES( $states, size, count, m$ )
    if  $count > 0$  then return DECIDE( $states$ )

```

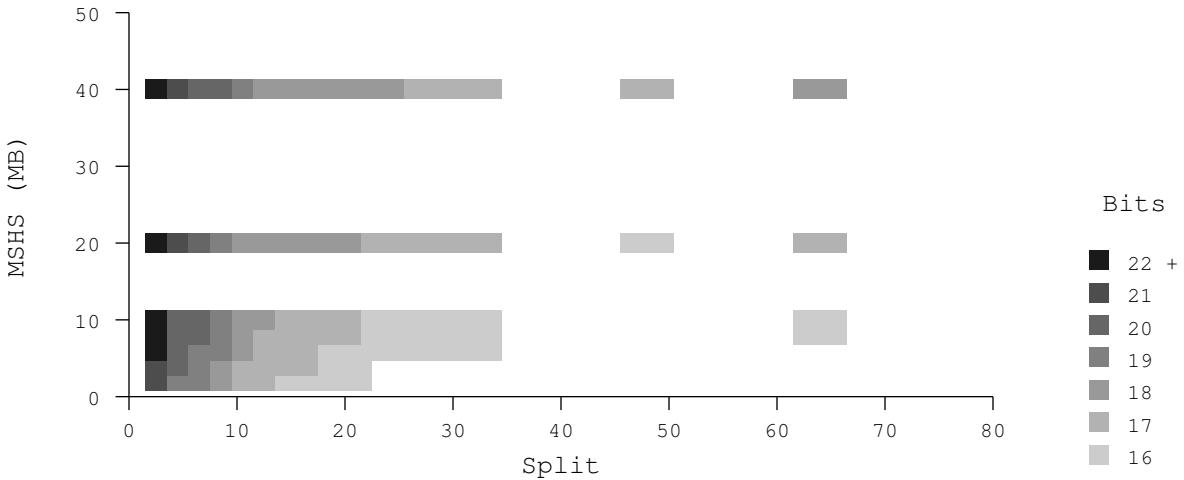


Figure 4: Residual entropy for different value combinations of the split and MSHS parameters (lighter is better)

A crucial insight is that when an allocation succeeds, all states that have a highest maximum value which cannot accommodate the successful allocation size are necessarily impossible and are pruned from the set of active states. This serves to contain the number of active states and somewhat ameliorate the combinatorial behavior of the forking mode.

The generalized algorithm (Algorithm 2) treats all cases (single or multiple states, the hmv is unique or has multiplicity n) uniformly. First, it considers the maximum values of the holes within a single state. Multiplicity is established within each state. Following that, the holes are sorted in descending order based on their maximum values and then again in descending order with regard to the multiplicity of each maximum value in the state it originated from. The number of times an allocation is repeated is determined by the multiplicity of the topmost hole; the descent takes place between that hole and the next hole of a different maximum value.

Intuitively, if we have exactly two states, S_1, S_2 with unique hmv values hmv_1, hmv_2 and that $hmv_1 = hmv_2$. Then we need try an allocation which can only be satisfied by the maximally-sized hole of either state only once; if the allocation succeeds, both states are updated and remain valid. Conversely, if hmv_1 has multiplicity two and hmv_2 only one, we need to try the allocation two times. If both allocations succeed, S_1 gets updated accordingly and S_2 gets dropped as invalid; if only one allocation succeeds, S_1 is replaced by two new states whereas S_2 is adjusted and remains live.

C Further evaluation of the PAP-only attack

When considering Figure 4 visualizes the residual accuracy (in bits) in the size of the largest hole next to the hidden object. The corresponding figure for the smaller hole appears almost identical and is omitted for brevity.

For the split factor, we investigated values ranging from 4 to 64, specifically 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 48 and 64. Guided by a sampling of typical applications on workstations and servers, we considered the following upper bounds for the sizes of untracked holes: 2MiB, 4MiB, 6MiB, 8MiB, 20MiB and 40MiB.

Our evaluation of the PAP in weakening ASLR’s protection of a hidden object involved seeding the algorithm with an initial state consisting of 4 hole descriptions. Specifically, we included the maximum possible values for the holes resulting from the placement of the hidden object at a random address within hole A. Hence, our tracked holes were Large, Small, B and C.

Analyzing Figure 4, we notice the tendency for larger split values to result in lower uncertainty. This tendency is not consistent and may vary with the MSHS; for example, split=48 outperforms split=64 for larger MSHS and is overall the best choice at larger MSHS values.

The split factor of 4 is by far the worst choice. It should be mentioned that the runtime performance deteriorates to the point of being impractical when using 2 as the split factor (for reasons expounded on in B, which led us to exclude it from the parameter exploration).

What Cannot be Read, Cannot be Leveraged? Revisiting Assumptions of JIT-ROP Defenses

Giorgi Maisuradze

CISPA, Saarland University

Saarland Informatics Campus

gmaisura@mmci.uni-saarland.de

Michael Backes

CISPA, Saarland University

Saarland Informatics Campus

backes@mpi-sws.org

Christian Rossow

CISPA, Saarland University

Saarland Informatics Campus

crossow@mmci.uni-saarland.de

Abstract

Despite numerous attempts to mitigate code-reuse attacks, Return-Oriented Programming (ROP) is still at the core of exploiting memory corruption vulnerabilities. Most notably, in JIT-ROP, an attacker dynamically searches for suitable gadgets in executable code pages, even if they have been randomized. JIT-ROP seemingly requires that (i) code is *readable* (to find gadgets at run time) and (ii) executable (to mount the overall attack). As a response, Execute-no-Read (XnR) schemes have been proposed to revoke the read privilege of code, such that an adversary can no longer inspect the code after fine-grained code randomizations have been applied.

We revisit these “inherent” requirements for mounting JIT-ROP attacks. We show that JIT-ROP attacks can be mounted without ever reading any code fragments, but instead by *injecting* predictable gadgets via a JIT compiler by carefully triggering useful displacement values in control flow instructions. We show that defenses deployed in all major browsers (Chrome, MS IE, Firefox) do not protect against such gadgets, nor do the current XnR implementations protect against code injection attacks. To extend XnR’s guarantees against JIT-compiled gadgets, we propose a defense that replaces potentially dangerous direct control flow instructions with indirect ones at an overall performance overhead of less than 2% and a code-size overhead of 26% on average.

1 Introduction

Code-reuse attacks, such as Return-Oriented Programming (ROP), enable an attacker to bypass Execute-XOR-Write (X^W) policies by suitably chaining existing small code fragments (so-called *gadgets*). One of the most prominently explored concepts to defend against such attacks involves randomizing programs so that an attacker can no longer reliably identify and chain such gadgets, whether by code transformations [19, 6, 14], data region

hardening [5, 23], or whole address space randomization [4]. However, a novel class of attacks, dubbed JIT-ROP, allows for code reuse even for such diversified programs [31]. JIT-ROP leverages a memory disclosure vulnerability in combination with a scripting environment—which is part of all modern browsers—to read existing code parts, notably *after* they were randomized. Once the code has been read (e.g., using a memory disclosure vulnerability), an attacker can dynamically discover and chain gadgets for conventional code-reuse attacks.

Mounting a successful JIT-ROP attack seemingly requires the ability to (i) read code fragments and identify suitable gadgets (otherwise the adversary would not know what to combine) and to (ii) execute them (so that the overall attack can be mounted). The recently proposed Execute-no-Read (XnR) schemes [2, 11, 12] consequently strive to eliminate JIT-ROP attacks by ensuring that executable code is *non-readable*, i.e., marking code sections as executable-only while explicitly removing the read privilege. Hence an adversary can no longer inspect the code after fine-grained code randomization techniques have been applied and should thus fail to identify suitable gadgets. As a more pointed statement: what cannot be read, cannot be leveraged.

Our contributions: In this paper, we carefully revisit these seemingly inherent requirements for mounting a successful JIT-ROP attack. As our overall result, we show that JIT-ROP attacks can often be mounted without ever reading any code fragments, but by instead injecting *arbitrary, predictable* gadgets via a JIT compiler and by subsequently assembling them to suitable ROP chains without reading any code pages.

As a starting point, we show how to obtain expressive unaligned gadgets by encoding specially-crafted constants in instructions. Prior research has already shown that *explicit* constants in JavaScript statements, e.g., in assignment statements like $x = 0x12345678$, can be used to generate unaligned gadgets [1]. Browsers started to fix such vulnerabilities, e.g., by blinding explicit con-

stants (i.e., XOR-ing them with a secret key), and/or by applying fine-grained code randomization techniques (cf. Athanasakis *et al.* [1]). We show that *implicit* constants in JIT-compiled code can be exploited in a similar manner, and are hence far more dangerous in the JIT-ROP setting than commonly believed. To this end, we generate JavaScript code that emits specific offsets in relative jumps/calls in the JIT-compiled code. We show that both relative jumps and relative calls can be used to encode attacker-controllable values in an instruction’s displacement field. These values can later be used as unaligned gadgets, i.e., an attacker that controls the jump or call destination (or source) can predict the displacement and thereby generate deterministic gadgets on-the-fly, without the need to ever read them before use. We demonstrate the impact of our attack by injecting almost arbitrary two- or three-byte-wide gadgets, which enable an attacker to perform arbitrary system calls, or, more generally, obtain a Turing-complete instruction set. We show that all major browsers (Chrome, Internet Explorer, Firefox) are susceptible to this attack, even if code randomization schemes such as NOP insertion (like in Internet Explorer) are in place.

The ability to create controllable JIT-compiled code enables an adversary to conveniently assemble ROP chains without the requirement to ever read code. This challenges current XnR instantiations in that code does not have to be readable to be useful for ROP chains, highlighting the need to complement XnR with effective code pointer hiding and/or code randomization schemes also in JIT-compiled code. Unless XnR implementations additionally protect JIT-compiled code, they do not prevent attackers from reusing predictable attacker-generated gadgets, and hence from mounting JIT-ROP attacks. We stress that a complete XnR implementation that offered holistic code coverage (i.e., hiding code-pointers also in JIT-compiled code) may also be effective against our attack. However, maintaining XnR’s guarantees also for JIT-compiled and attacker-controlled code imposes additional challenges in practical settings: First, fine-grained code randomization schemes that are implemented in XnR do not add security against implicit constants, and they hence make gadget emissions, proposed in this paper, possible. In particular, the widely deployed concepts of register renaming and instruction reordering do not affect our proposed unaligned gadgets. Moreover, fine-grained code randomization techniques commonly deployed in browsers (as NOP insertion in IE) are not sufficient either, as the attacker can test the validity of its gadgets before using them. Second, the lack of code pointer hiding in JIT-compiled code in current XnR instantiations constitutes an additional vector of attack, since adversaries can then still leverage our attack to encode constants in relative calls.

We finally explored how to extend XnR’s guarantees against implicit constants in JIT-compiled code. One option would be to extend the use of call trampolines in XnR schemes also to JIT compilers, as suggested by Crane *et al.* [11, 12]. However, this will replace existing direct calls with direct jumps to trampolines, which also encode implicit constants. Furthermore, trampolines will introduce new relative offsets in their direct call instructions. As the locations of trampolines are not hidden (e.g., they can be revealed by reading the return address on stack), in the presence of an unprotected code pointer, the attacker will be able to predict encoded constants by leaking either the caller or the callee address. As an orthogonal alternative, in this paper we propose to (i) replace relative addressing with indirect calls and (ii) blind (i.e., reliably obfuscate) all explicit constants used to prepare the indirect calls. We implement our defense in V8, the JavaScript engine of Chrome, and show that our proposal imposes less than 2% performance and 26% code size overhead, while effectively preventing the attacks described in this paper.

The summarized contributions of our paper are:

- We present a novel class of attacks that encode ROP gadgets in implicit constants of JIT-compiled code. We thereby show that reading code fragments is *not* necessarily a prerequisite for assembling useful gadgets in order to mount a JIT-ROP attack.
- We demonstrate that all three major browsers (Chrome, Internet Explorer, Firefox) are susceptible to our proposed attack.
- We discuss potential shortcomings when using XnR to protect JIT-compiled code. We show that the underlying assumptions that XnR schemes build upon (such as code randomization) have to be carefully evaluated in the presence of JIT-compiled code.
- We implement a defense in V8 that replaces relative calls/jumps with indirect control flow instructions. This effectively prevents the attack proposed in this paper by removing dangerous implicit constants, exhibiting a performance overhead of 2% and a code size overhead of 26%.

The remainder of this paper is structured as follows. Section 2 provides background information on code-reuse attacks. Section 3 describes our threat model. Section 4 introduces the fundamentals of our attack and demonstrates its efficacy against three major browsers. Section 5 introduces an efficient defense against our attack. Section 6 discusses the implications of our work. Section 7 describes related work and Section 8 concludes the paper with a summary of our findings.

2 Background

We will use this section to provide background information on code-reuse attacks. We start by explaining ROP, and then provide insights on JIT-ROP, which collects code on-the-fly and thus evades existing randomization schemes like ASLR. Finally, we describe Execute-no-Read (XnR), a new defensive scheme that aims to protect against code-reuse attacks (including JIT-ROP).

2.1 Return Oriented Programming (ROP)

ROP has emerged since the wide deployment of Data Execution Prevention (DEP), which is a defense technique against regular stack overflow vulnerabilities. DEP, making the writable regions of the memory non-executable, forbids the attacker to execute the shellcode directly on the stack. As a response, attackers switched to code-reuse attacks, in which they execute *existing* code instead of injecting new code. ROP, proposed by Shacham [29], is a generalized version of the `ret-to-libc` attack [22], which redirects the control flow of the program to existing code, such as the program’s code or imported functions (e.g., in `libc`). In ROP, an attacker uses short instruction sequences (called gadgets) ending with a control flow instruction (e.g., `ret`). Return instructions are used to chain multiple gadgets together by providing their addresses as the return values on the stack. Checkoway *et al.* [8] showed that it is possible to launch ROP attacks without using return instructions, i.e., via leveraging other control flow changing instructions such as indirect jumps or calls.

Code-reuse remains a popular attack technique and has triggered a variety of defensive schemes. Most prominent, and deployed in most operating systems, is Address Space Layout Randomization (ASLR). ASLR randomizes the base addresses of memory segments and prevents an attacker from predicting the addresses of gadgets. Although ASLR is effective for pre-computed gadget chains, ASLR has known shortcomings in that it only randomizes base addresses and is too coarse-grained. An attacker can thus reveal the memory layout of an entire ASLR-protected segment with a single leaked pointer. To address this problem, fine-grained ASLR randomization schemes have been proposed that add randomness inside the segment [16, 20, 25] (we refer the reader to Larsen’s survey [21]).

2.2 JIT-ROP

To counter ASLR, Snow *et al.* proposed a new attack technique, called *just-in-time* code reuse (JIT-ROP) [31]. By leveraging the fact that an adversary is able to read randomized code sections, JIT-ROP undermines fine-

grained ASLR schemes. JIT-ROP is based on the following assumptions: (i) a memory disclosure vulnerability, allowing the attacker to read data at arbitrary locations, (ii) at least one control flow vulnerability, (iii) a scripting environment running code provided by the attacker. The basic idea of the JIT-ROP is the following:

- (J1) Repeatedly using the memory disclosure vulnerability, the attacker follows the code pointers in the memory to read as many code pages as possible.
- (J2) From the read code pages, JIT-ROP extracts gadgets (e.g., Load, Store, Jump, Move) and collects useful API function calls (e.g., `LoadLibrary`, `GetProcAddress`).
- (J3) Given the gadgets and API functions, the JIT-ROP framework takes an exploit, written in a high-level language, as an input and compiles it to a chain of gadgets and function calls that perform a code-reuse attack.
- (J4) Finally, the control flow vulnerability is used to jump to the beginning of the compiled gadget chain.

JIT-ROP demonstrates that an adversary may be able to run code-reuse attacks even in the case of fine-grained ASLR or code randomization, as she can read the code and function pointers *after* they have been randomized.

2.3 Execute-no-Read (XnR)

In an attempt to close the security weakness that JIT-ROP has demonstrated, researchers suggest marking code sections as *non-readable*. Such Execute-no-Read (XnR) schemes were proposed by Backes *et al.* [2], and were strengthened by Crane *et al.* [11, 12] shortly thereafter. The common goal is to prevent step (J1) of the JIT-ROP attack, as the attacker can no longer dynamically search for gadgets in non-readable code sections.

XnR: Lacking support for XnR pages in the current hardware, Backes *et al.* implemented XnR in software by marking code pages as *non-present* and checking the permissions inside a custom pagefault handler [2]. To increase the efficiency of this scheme, the authors propose to leave a window of N pages present. This exposes a few readable pages to the attacker, but prevents her from reading *arbitrary* code pages. As the authors suggest, at low window size ($N = 3$), the likelihood that an attacker can leverage code-reuse attacks using only the present code pages is negligible.

Readactor(++): Crane *et al.* suggested Readactor [11] and Readactor++ [12], both of which leverage hardware support to realize XnR. The authors suggest using Extended Page Tables (EPT), which were introduced recently to the hardware to assist virtualized environments. While regular page tables translate virtual addresses into physical ones, EPTs add another layer of indirection

and translate physical addresses of a VM to physical addresses of the host. EPTs allow marking pages as (non-)readable, (non-)writable, or (non-)executable, allowing enforcement of XnR in hardware. In addition, Readactor(++) hides code pointers by creating trampolines, and replacing all code pointers in readable memory with trampoline pointers. The underlying assumption of Readactor(++) is that fine-grained code diversification techniques are in place, such as function permutation, register allocation randomization, and callee-saved register save slot reordering.

Despite the fact that Readactor hid code pointers, the layout of some function tables (e.g., import tables or vtables) stayed the same. This allows an adversary to guess and reuse the function pointers from these tables [27]. Readactor++ fixed this issue by randomizing these tables (to get rid of the predictable layout) and randomly injecting pointers to illegal code (to forbid function fingerprinting by executing it).

Alternative XnR Designs: Gionta *et al.* [15] proposed HideM, which, using a split TLB technology, differentiates between memory accesses and only allows instruction fetches to access code pages. Further, HideM considers the data in the executable memory pages that need to be read and uses read policies to guarantee their security. However, although HideM might be used to enforce non-readable code, it highly depends on the hardware support (e.g., split TLB). Furthermore, HideM does not hide code pointers. In addition, Pereira *et al.* [26] designed a technique similar to Readactor(++) that aims to get non-readable code for mobile devices in ARM. One of the advantages of their approach, called Leakage-Resilient Layout Randomization (LR²), is that it is implemented in software and does not require the underlying hardware support. LR² achieves this by splitting the memory space in half (into code and data pages) and instrumenting load instructions to forbid the attacker to read code. LR² also optimizes the use of trampolines by creating only a single trampoline for each callee instead of encoding one for each callee-caller pair.

Summarizing XnR: Even though the current XnR implementations mark JIT-compiled code as non-readable, existing prototypes allow to leak JIT code pointers via JIT-compiled code. We will show that an adversary that controls the JavaScript code can still run code-reuse attacks on the code generated by the JIT compilers.

2.4 JIT-Compiled Gadgets

JIT compilation remains a major challenge for XnR implementations. During JIT compilation, the JIT engine (e.g., of a browser) compiles JavaScript code into assembly instructions to optimize performance. This is done by converting each JavaScript statement into a sequence

of corresponding assembly instructions, making the code output of the JIT compiler predictable. The deterministic JIT compilation allows an attacker to influence the code output by controlling the JavaScript code. For example, Blazakis [7] and Athanasakis *et al.* [1] propose to craft special JavaScript statements that JIT-compile into gadgets. Consider a statement with an immediate value, such as the assignment `var a=0x90909090`. The JIT engine will compile this into a sequence of assembly instructions, one of them being a `mov eax,0x90909090` instruction that encodes the attacker-chosen immediate value. After the compilation, the attacker can jump in the middle of the instruction and use the bytes of the immediate value as an unaligned gadget, such as four consecutive `nop` instructions in our simple example.

JIT Compiler Defenses: Modern JavaScript compilers prevent such unaligned gadgets by *constant blinding*. Instead of directly emitting constants in native code, compilers XOR them with randomly generated keys, making the resulting constants unpredictable. After constant blinding, the aforementioned JavaScript statement will be compiled to the following assembly instructions, effectively removing the attacker-controlled constant:

```
mov eax, (0x90909090 ⊕ KEY)
xor eax, KEY
```

For performance reasons, modern browsers only blind large constants. For example, Chrome and IE blind constants containing three or more bytes, giving the attacker a chance to emit arbitrary two-byte gadgets. Athanasakis *et al.* [1] demonstrated that two-byte gadgets are sufficient to mount a successful ROP attack, provided that (i) code sections are readable, and (ii) available gadgets happen to be followed by a `ret` instruction.

While constant blinding protects against such gadget emissions, as we will show, it does not protect against our novel form of JIT-compiled *implicit constant* gadgets.

3 Assumptions

We now describe our assumptions that we follow throughout this paper, detailing a threat model and discussing defenses that we assume are in place on the target system. These assumptions are in accordance with the recently proposed defense mechanisms against JIT-ROP, such as XnR [2] and Readactor [11].

3.1 Defense Techniques

We assume that the following defense mechanisms of the operating systems and the target application are in place:

- **Non-Executable Data:** Data Execution Policy (DEP) is enabled on the target system, e.g., by us-

- ing the NX-bit support of the hardware, marking writable memory pages non-executable.
- **Address Space Layout Randomization:** The target system deploys base address randomization techniques such as ASLR, i.e., the attacker cannot predict the location of a page without a memory disclosure vulnerability. In addition, we assume popular fine-grained ASLR schemes [20, 33, 16, 25, 17], as suggested by current XnR implementations [11, 12], are applied on the executable, libraries, and JIT-compiled code.
 - **Non-Readable Code:** We assume that all code segments are non-readable, with this being either software- [2] or hardware-enforced [11, 12], notably also assuming that JIT-compiled code is non-readable.
 - **Hidden Code Pointers:** We assume that all code pointers, except for JIT-compiled ones, are present but anonymized, e.g., via pointer indirections such as trampolines proposed by Readactor. Note that, as mentioned by Crane *et al.*, Readactor(++) could be extended to also hide code pointers in JIT-compiled code. However, there is no implementation that shows this, neither is the performance impact of such a scheme clear. In addition, having the compiler running in the same process as the attacker might give the adversary the ability to read code pointers *during* the compilation process. We thus believe that hiding all possible (direct or indirect) code pointers is a challenging task and the attacker might still be able to leak the required function addresses.
 - **JIT Hardening:** We assume modern JIT defenses such as randomized JIT pages, constant blinding, and guard pages (i.e., putting an unmapped page between mapped ones). In our attack, for simplicity, we assume that sandboxing is either disabled or can be bypassed via additional vulnerabilities. In addition, assessing the security of Control Flow Integrity (CFI) defenses in JIT compilers is out of scope of this paper, as our core contribution is to show that an attacker can *inject* gadgets, and not to discuss the actual process of diverting control flow. Instead, we demonstrate the threat of attacker-controlled code emitted by the JIT compiler.

3.2 Threat Model

In the following, we enumerate our assumptions about the attacker. This model is consistent with the threat model of previous attacks such as JIT-ROP [31] and with the XnR-based defense schemes.

- **Memory Disclosure Vulnerability:** We assume that the target program has a memory disclosure vulnerability, which can be exploited repeatedly by the attacker to disclose the *readable* memory space (i.e., we can read data, but cannot read code).
- **Control-Flow Diversion:** We assume that the target program has a control-flow vulnerability, allowing the attacker to divert the control flow to an arbitrary location. Note that this by itself does not allow the attacker to exploit the program, given the lack of ROP gadgets due to fine-grained ASLR and XnR.
- **JavaScript Environment:** We assume that the vulnerable process has a scripting environment supporting JIT compilation, for which the attacker can generate arbitrary JavaScript code. This is common for victims that use a browser to visit an attacker-controlled web site. Similarly, it applies to other programs such as PDF readers.

4 JIT-Compiled Displacement Gadgets

In this section, we discuss how an attacker can induce *new* JIT-compiled gadgets by crafting special JavaScript code. Intuitively, we show that an attacker can generate predictable JIT-compiled code such that she can reuse the code without searching for it. We introduce new techniques to trigger predictable gadgets that all modern JavaScript engines happen to generate. We demonstrate that an attacker can create and use almost arbitrary x86/x64 gadgets in modern browsers and their corresponding JavaScript engines, such as Google Chrome (V8), MS Internet Explorer (Chakra), and Mozilla Firefox (SpiderMonkey).

We introduce two techniques to emit gadgets via implicit constants. First, in Section 4.1, we leverage JavaScript’s control flow instructions and emit conditional jumps (such as `je 0x123456`) that may encode dangerous offsets. Second, in Section 4.3, we show how an attacker can leverage offsets in direct calls, such as `call 0x123456`, to create gadgets.

4.1 Conditional Jump Gadgets

Our first target is to turn offsets encoded in conditional jumps (in JIT-compiled code) into gadgets. To this end, we use JavaScript statements, such as conditionals (`if/else`) or loops (`for/while`), that are compiled to conditional jumps. Figure 1(A) shows an example. In `js_gadget`, the body of the `if` statement contains a variable-length JavaScript code. After compilation, the `if` statement is converted to a sequence of assembly instructions containing a conditional jump, which, depending on the branch condition, either jumps over the body or falls through (e.g., `je <if_body_size>`). By varying

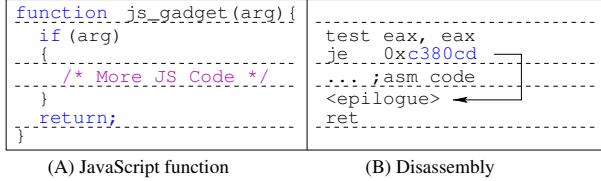


Figure 1: JavaScript function js_gadget and its corresponding disassembly

the code size inside the `if` body, we change the jump distance and thus the value encoded in the displacement field of the jump instruction in the compiled code. For example, if we aim for a `int 0x80;ret (0xcd80c3)` gadget, we have to fill the body of the `if` statement with JavaScript code that is compiled to `0xc380cd` bytes. The size of the JIT-compiled code for each JavaScript statement is fixed by the corresponding JIT compiler, and thus an attacker can precisely generate code of any arbitrary length. Note that the bytes of the size and the emitted gadget are mirrored because of the little-endian format used in x86/x64 architectures. The compiled version of `js_gadget` is shown in Figure 1(B).

Emitting such three-byte gadgets requires large portions of JavaScript code. In case the malicious JavaScript code has to be loaded via the Internet, this might drastically increase the time required for all gadgets to be in place. An attacker could overcome this limitation by utilizing the `eval` function. Instead of having ready-made JavaScript code, we thus use a function that constructs and emits all required gadgets on-the-fly. Such a script to dynamically generate arbitrary gadgets occupies less than one kilobyte.

In a naïve attack instantiation, each additional gadget will increase the overall code size. To counter this potential limitation, we can also embed smaller gadgets into the bigger ones by stacking `if` statements inside the body of another `if` statement, ideally reducing the size of the JavaScript code to the size of the biggest gadget.

Computing addresses of JavaScript functions: In order to use generated gadgets, we have to compute their addresses. We start by revealing the address of the JIT-compiled JavaScript function, which contains emitted gadgets, by employing a memory disclosure vulnerability. We can do this, for example, by passing the function as a parameter to another one, thus pushing its value on the stack. Afterwards, in the callee, we read the stack, revealing the pointer to the function's JavaScript object, which contains the code pointer to the actual (JIT-compiled) function. Note that here we assume that we know the location of the stack. This can be done by chasing the data pointers in the readable memory, until we find a pointer pointing to the stack.

Because of the predictable code output of JIT compil-

ers, we know the offsets inside the JIT-compiled function, at which conditional jumps will be emitted and can thus compute the addresses of emitted gadgets.

4.2 Conditional Jump Gadgets in Browsers

We tested this technique against three modern browsers: Chrome 33 (32-bit)/Chrome 51(64-bit), Firefox 42 (64-bit) and IE 11 (64-bit with 32-bit JavaScript engine). There are some differences that need to be taken into account for each of them. For example, Chrome compiles JavaScript functions the first time they are called, while Firefox and IE interpret them a few times until they are called too often (e.g., around 50 times for IE and 10 times for Firefox) and only then JIT-compile the JavaScript code. Therefore, to trigger the compilation we just call the function multiple times and then wait until it is compiled (which takes a few milliseconds).

Chrome: As each browser has its own JIT compiler, an attacker has to vary the JavaScript code to fill the exact number of bytes in the if body. This is just a matter of finding a mapping between JavaScript statements and the number of bytes of their JIT-compiled equivalent. We will demonstrate this by emitting a system call gadget (`int 0x80;ret`) in 32-bit Chrome. To this end, we need to emit `0xcd80c3`, i.e., we need to fill the if body with JavaScript code that is JIT-compiled to `0xc380cd` bytes. We use the following two JavaScript statements:

S1: $v=v1+v2$, compiling to 0x10 bytes, and
S2: $v=0x01010101$, compiling to 0xd bytes.

By combining these two statements, we can generate arbitrary gadgets. In our case, we use S1 0x0c 38 0c times (resulting in 0xc3 80 c0 bytes) and S2 once—summing up to 0xc3 80 cd, our desired gadget.

Note that the JavaScript statement that compiles to 0x10 bytes allows us to control each hex digit of the emitted jump distance except the last one (i.e., the least significant half-byte of the gadgets' first byte). Moreover, any JavaScript statement that compiles to an odd number of bytes allows us to control the least significant half-byte of the distance. Combining these two properties, we can generate any gadget by using these two selected JavaScript statements multiple times.

The sizes of JIT-compiled JavaScript statements differ in 64- and 32-bit versions of Chrome. In 64-bit Chrome we replace S1 with `v=v`, which is compiled to 0x10 bytes. Note, however, that even though the size of S2 in 64-bit is also changed to 0x1b bytes, we can still use it because it is compiled to an odd number of bytes.

Firefox: To generate arbitrary gadgets for Firefox, we choose the following two statements:

S1: $v=v$, compiled to 8 bytes (two of them to $0x10$), and
S2: $v+=0x1$, compiled to 0x21 bytes.

```

0x0000: call FUN_1      ; 0xe8fb1f0000
0x0005: call FUN_1      ; 0xe8f61f0000
0x000a: ...
0x2000: push ebp        ; 0x5d(@FUN_1)

```

Figure 2: Direct call

IE: IE deploys JIT-hardening mechanisms that go beyond the protections in Chrome and Firefox. IE (i) has a size limit on code segments generated by the JIT compiler, and (ii) randomly inserts NOPs (i.e., instructions that do not change the program state) in JIT-compiled code. Because of (i), we can only emit two-byte gadgets. Due to (ii), these gadgets are then further modified by inserting NOPs inside the if body and, thus, changing the value emitted in the conditional jump. The latter technique is similar to librando [17]. Nevertheless, even with these defenses in place, we can still emit arbitrary two-byte gadgets by measuring the size of the emitted code at run time. We will describe this attack in Section 4.4 in the discussion about Internet Explorer.

4.3 Direct Call Gadgets

We found that conditional jumps are not the only instructions to embed implicit constants that can be indirectly controlled by the attacker. Direct calls (e.g., `call 0x1234560`) are another example of such instructions. In our second approach, we leverage the JavaScript statements that are compiled to instructions containing direct calls.

Direct call constants: Direct calls in x86/x64 change the execution flow of the program by modifying the instruction pointer (`eip/rip`). The constant encoded in a direct call instruction represents a relative address of the callee. That is, the call instruction’s displacement field contains the distance between the addresses of the instruction following the call and the callee. Therefore, any two direct call instructions to the same function will encode different constants. For example, in Figure 2, there are two consecutive calls to the function `FUN_1` (at address `0x2000`). The constant encoded in the first call denotes the distance between `FUN_1` and the instruction following the call (i.e., the second call at `0x05`). Therefore, its value is $0x2000 - 0x5 = 0x1ff5$, which is `0xfb1f0000` encoded in little-endian.

In the example above, the difference between two consecutive direct call constants is `0x5` (the size of a direct call instruction). In general, the difference is equal to the size of the instructions between two consecutive calls. In our case, we want to use JavaScript statements to emit direct calls in the JIT-compiled code. Therefore, the difference between the constants will be the size of the in-

```

function js_call_gadget() {
    asm_call(); /* emits a call */
    asm_call();
    /* ... (many asm_call() statements) */
    asm_call();
}

```

Listing 1: JavaScript function `js_call_gadget`

structions in which the JavaScript statement is compiled.

To generalize this attack vector, we aim for a JavaScript function similar to `js_call_gadget` (Listing 1). The `asm_call()` statement is a placeholder for any JavaScript statement (not necessarily a function call) that is compiled into a sequence of instructions containing a direct call. The exact statement that replaces the placeholder depends on the target browser.

Finding callee address: Let our goal be to emit a three-byte gadget and fix its third byte to `0xc3` (`ret`). To calculate the constant encoded in the displacement field of a direct call instruction, we have to know the addresses of the call instruction and its destination. The destinations of the emitted call instructions that we have encountered are either helper functions (e.g., inline caches generated by V8) or built-in functions (such as `Math.random` or `String.substring`). The helper functions are JIT-compiled by V8 as regular functions. We can leak their addresses either by stack reading (e.g., by leaking the return address put there by the call instruction inside the helper function), or by reading the V8’s heap, where all the references of compiled helper functions are stored. In IE, the built-in functions are located in libraries and thus are randomized via fine-grained ASLR schemes [12, 11, 17]. However, their corresponding JavaScript objects (e.g., `Math.random`) contain the code pointer to the function. Knowing the structure of these JavaScript objects, which are not randomized according to our assumptions, we can get the addresses of built-in functions via a memory disclosure vulnerability. Note that after code pointer hiding, the addresses that the attacker leaks from these JavaScript objects will be the addresses of the trampolines and not the actual functions. Nevertheless, offsets, encoded in call (or jump) instructions, will also be computed relative to the trampolines and thus can be used for calculating emitted constants.

Emitting call instructions: Knowing the address of the callee, the next step is to emit direct call instructions at the correct distance. Given that we cannot influence the address where the function will be compiled, we have to acquire sufficiently large code space to cover all three-byte distances to the callee. To this end, we create a JavaScript function that spans `0x1 00 00 00` bytes after JIT compilation and consists of JavaScript statements emitting direct calls. More precisely, we require the distance between the first and the last emitted direct call

instructions to be at least 0x100000 bytes. This way, regardless of where our function is allocated, we will be guaranteed that it covers all possible three-byte distances from the callee, allowing us to emit arbitrary three-byte gadgets by carefully placing direct call instructions.

Emitting required gadgets: Creating such a large function (16 MB) emits many three-byte gadgets, and also covers all two-byte gadgets. For example, if we have a JavaScript statement that generates a call instruction and is compiled to 0x10 bytes of native code, we can create a big function containing this statement 0x100000 times. The compiled function will have 0x100000 direct call instructions 0x10 bytes apart. If we consider the least significant three bytes of the emitted displacement fields of these direct calls, they will have the following form: $0x*Y***$, where * denotes any hexadecimal digit [0-f] and Y is a constant, which encodes the least significant half-bytes of emitted values.

Because of the little-endian format used in x86/x64 architectures, Y is part of the first byte of emitted gadgets. Therefore, to emit three-byte gadgets, we must be able to set Y accordingly. To this end, we modify the value of Y by varying the size of the instructions before the first direct call. That is, we find any JavaScript statement that compiles to an odd number of bytes, and then use it up to 15 times to get any out of all possible 16 half-bytes. For example, if the least significant half-byte of the call instruction is 0x0 and we want to make it 0xd, and we have a JavaScript statement that compiles to an odd number of bytes (e.g., `i+=1` in 32-bit Chrome, 0x13 bytes), we use this statement 15 times ($0x13*15=0x11d$).

Computing addresses of emitted gadgets: Assuming that the address of the first call instruction in our function is F_{call} , and the address of the callee is F_{dest} , we can compute three bytes of the displacement field of the first call instruction by $C_1 = F_{dest} - (F_{call} + 5) \bmod 2^{24}$. If the required gadget is G, then we can compute the distance (dist) between the call instruction, emitting G, and the first call instruction: $dist = C_1 - G \bmod 2^{24}$. Using dist, we can calculate the address of the call instruction emitting G ($F_{call} + dist$), and therefore the address of the gadget (G_{addr}) which is located 1 byte after the call: $G_{addr} = (F_{call} + dist) + 1$.

4.4 Direct Call Gadgets in Browsers

We will next discuss techniques that we use to instantiate the attack in three popular browsers.

Firefox: Emission of direct call gadgets is not possible in Firefox, as the baseline JIT compiler of Firefox does not emit direct calls. Although the optimizing JIT compiler of Firefox emits direct calls, e.g., when compiling regular expressions, it only optimizes JavaScript functions after they have been executed more than 1,000 times. Trig-

gering the optimizing compiler on the large functions (as required for our attack) thus makes our attack impractical against Firefox.

Chrome: Chrome compiles most JavaScript statements to direct calls. Consequently, we have a large selection of JavaScript statements with varying post-compilation sizes. We use a statement that is compiled to 0x10 bytes of assembly code (e.g., `i=i+j` for 32-bit Chrome). For demonstration purposes, we aim to emit a system call gadget (`int 0x80; ret`), implicitly also revealing all two-byte gadgets. To this end, we create a function shown in Figure 3(A). The function starts with a sequence of JavaScript statements that align the first call instruction to 0xe. After this, the emitted call distances will be calculated relative to 0x3, i.e., $(0xe+0x5) \bmod 0x10$, where a direct call is 0x5 bytes large. Considering that the callee is at least half-byte aligned, the lower half-byte of all emitted gadgets' first bytes will be 0xd, i.e., $(0x0-0x3) \bmod 0x10$. The alignment code is followed by a sequence of call-generating statements (e.g., `i=i+j`), each of which compiles to 0x10 byte-long code. The compiled `i=i+j` statement emits a call instruction at offset 0x*e (due to alignment) as shown in Figure 3(B). Generating a sequence of 0x100000 call instructions, we are guaranteed to have an `int 0x80; ret` gadget encoded into one of the call instruction constants (Figure 3(C)).

Note that the aforementioned technique is used in the 32-bit version of Chrome. For 64-bit, we use the `v++` statement, which is compiled to 0x20 bytes (instead of 0x10) and emits a `call` instruction. Having 0x20 bytes between `call` instructions changes the upper half of the least significant byte. For example, after aligning the least significant half-byte to 0xd via padding, the emitted first bytes will be either $0x\{0,2,4,6,8,a,c,e\}d$ or $0x\{1,3,5,7,9,b,d,f\}d$, depending on the initial value of the upper half of the least significant byte. We can modify this value to our liking by adding the `i=i` statement, which is compiled to 0x10 bytes, as padding.

Internet Explorer: For the two main reasons mentioned in Section 4.1 (code size limit and NOP insertion), emitting gadgets is harder in IE. The per-function code size limit forbids us to emit 0x100000 bytes of native code, which is required to span all possible third bytes of the constants encoded in call instructions. However, in the following, we describe how an attacker can still encode gadgets in direct calls even in IE.

Emitting calls at correct distance: IE still allows us to create many small functions. These functions will be distributed in the set of pages, each of them being 0x2000 bytes large. We thus allocate many functions (200 in our case), each of them being $\lesssim 0x1000$ bytes (i.e., two functions per page). Given the alignment (0x1000) and the size (0x2000) of the spanned code pages, each page will cover two third-bytes of

<pre>function js_call_gadget_v8() { var i, j; // Align calls to 0xe i = i + j; // i=i+j 0x3a86b times i = i + j; // i=i+j 0xc5793 times i = i + j; }</pre>	<pre>0x*0: ... 0x*7: ... 0x*e: call BINARY_OP_IC ;e8 8d07feff ... 0x*7: ... 0x*e: call BINARY_OP_IC ;e8 cd80c3ff ... 0x*7: ... 0x*e: call BINARY_OP_IC ;e8 8d07fefe</pre>	
(A) JavaScript function	(B) i=i+j direct calls	(C) Bytes emitted by direct calls

Figure 3: JavaScript function emitting gadgets via direct call constants

the absolute address completely (e.g., from 0x12₃₄0000 to 0x12₃₅ff ff). Considering that the callee is not aligned to the same boundary, direct calls emitted in these pages will have three distinct third bytes in their constants, only one of them covered completely. For example, if we assume the callee to be at address 0x12₃₄5670 and the page emitted at 0x01₇₀0000, only the call instructions located in the address range [0x01705670, 0x01715670] can emit complete three-byte constants, having 0xc3 as their third byte, i.e., constants from 0x12345670 - 0x01705670 = 0x10_{c4}0000 to 0x12345670 - 0x01715670 = 0x10_{c3}0000. On the other hand, the ranges [0x01700000, 0x01705670] and [0x01715670, 0x0171ff ff] cover only parts of the constants, with 0xc4 and 0xc2 as their third bytes.

After allocating the functions, we dynamically check their addresses to find the one with the correct distance from the callee (using the same technique as described at the end of Section 4.1), i.e., the one having the correct third byte in its direct call instruction’s displacement field. Allocating 200 JavaScript functions, each of them containing 0x10000 bytes, is inefficient, especially if the code has to be downloaded to the victim’s machine. Therefore, we use eval to spam IE’s code pages with the dynamically created functions. The problem with evaluated functions is that IE does not emit direct call instructions in them and uses indirect calls instead. Therefore, we use these functions only as temporary placeholders. Once we find any evaluated function at the correct place, we deallocate it to make its place available for the subsequently compiled functions. To deallocate a JavaScript function, we set null to all of its references and wait until the garbage collector removes it (typically within less than a second).

Verifying emitted gadgets: At first sight, IE’s NOP insertion conflicts with our assumption about the predictability of JIT-compiled code. With NOP insertion, and likewise with many other fine-grained code randomization schemes, we cannot guarantee that the call instruction, which is supposed to emit the gadget, ends up

```
function js_call_gadget_IE() {
    // Padding to correct address
    var i = Math.random(); // emit direct call
    check_address(<random cookie value>);
}
```

Listing 2: JavaScript function js_call_gadget_IE

at the correct address. However, because NOPs are inserted at random, compiling the same JavaScript function multiple times actually *increases* the chance that in one of the compiled versions, the call instruction ends up at the correct place.

Following our threat model, though, we cannot read executable code segments to verify if the compiled call instruction is at the desired place. As an alternative, we read the stack, as shown in Listing 2. In js_call_gadget_IE(), the statement i=Math.random() emits a direct call. We pad the beginning of the function with a few JavaScript statements to place i=Math.random() at approximately the correct address, such that the relative address would encode the desired gadget, accounting the randomness induced by NOP insertions. We then check the correctness of the position via check_address, a JavaScript function that reads the stack to find the return instruction pointer put there by the call instruction.

Using the leaked return address, we can calculate the address of the direct call instruction emitted by i=Math.random(), verify that it is at the correct place and if so, use it as a gadget. A simple implementation of check_address is shown in Listing 3, where it uses a memory disclosure vulnerability (mem_read in this case) to read the stack from some starting point (ESP_), until it finds its own parameter (cookie). The parameter is a random number, reducing the chance that multiple positions have the same value (note that this chance can be further reduced by using multiple random parameters). After finding the cookie on the stack and thus the address of the parameter, we know the exact offset

```

function check_address(cookie) {
    // ESP_: Any address on stack
    // NEEDED_ADDRESS: address where
    //                   call must reside
    while(mem_read(ESP_) != cookie)
        ESP_ -= 4; // Check next value
    // get return address from parameter
    var ret_ = mem_read(ESP_ - 0xC);
    // get call instruction address
    var call_addr = ret_ - 5;
    return call_addr == NEEDED_ADDRESS;
}

```

Listing 3: JavaScript function *check_address*

from the parameter’s address, and from there we can tell where the return address is located. Reading the return instruction pointer, we recover the address of the corresponding call instruction and verify that it is at the correct place (NEEDED_ADDRESS in this case). We can add another call to *check_address* before *i=Math.random()* to verify if NOPs are inserted between the emitted call instructions of *i=Math.random()* and *check_address()*. If both checks (*check_address*) succeed, we will be guaranteed that no NOPs were inserted.

IE summary: An attacker can evade both aforementioned defenses and emit three-byte gadgets even in IE. To demonstrate this, we first dynamically create many JavaScript functions to get the correct third byte (0xc3). After finding the function at the correct distance from the callee, we replace it with a special function, which, after compilation, emits direct call instructions and checks their positions. We trigger the recompilation of the latter function multiple times, until the checks are true, which means that the gadget is found. In our experiments, spamming the code pages with functions took approximately four seconds, and for most of the time, we found the correct third byte on the first try. Triggering the recompilation of a function takes the following steps: (i) Remove the included JavaScript file from the head of the HTML file, (ii) wait until the function gets removed by the garbage collector, and (iii) include the JavaScript file again and trigger the compilation of the same function. Each iteration of the above steps takes around 2 seconds, most of the delay coming from the second step (waiting for the garbage collector).

For two-byte gadgets, on the other hand, an attacker can discard the third byte. In this case, she can directly compile multiple functions at once, check the positions of emitted direct calls and use the ones with the required displacements.

4.4.1 Proof-of-Concept Gadget Generation

To demonstrate the practicality of the aforementioned gadget emitting techniques, we crafted a special JavaScript code for Chrome and IE, which generated the gadgets required for the exploit. The gadgets that we aimed to generate are the ones used by Athanasakis *et al.* Namely, the set of gadgets to load the registers with the arguments used in a system call and one for the system call itself. We created these gadgets in Chrome 51 (64 bit) and IE 11 (32 bit).

Chrome: For Chrome, we targeted for the following instructions: *pop r8*; *pop r9*; *pop rcx*; *pop rdx* (to prepare the system call arguments) and *int 0x80* (to execute the system call). Being able to emit three-byte gadgets, we encoded these instructions into the following gadgets:

<i>pop r8, ret ; 4158c3</i>
<i>pop r9, ret ; 4159c3</i>
<i>pop rcx, ret ; 59c3</i>
<i>pop rdx, ret ; 5ac3</i>
<i>int 0x80, ret ; cd80c3</i>

We used both our proposed techniques for the emission of these gadgets. We generated a system call gadget via direct calls. First, we created a string representation of a JavaScript function containing *0x80000 j++* statements (*j++* takes *0x20* bytes), then we created a JavaScript function from it via *eval*, and finally we compiled it by calling the generated function. This gave us a system call gadget, together with all possible two-byte gadgets, hence also covering *pop rcx* and *pop rdx*.

For the generation of *pop r8* and *pop r9* gadgets, we used cascaded *if* statements (also created with *eval*). The JavaScript function generating the aforementioned gadgets is shown in Listing 4. As gadgets *pop r8* and *pop r9* differ by *0x100*, their corresponding *if* statements also have to be *0x100* bytes apart. Note however that in the first *if* body (F1), we add *0xed* bytes to fill up the space instead of *0x100*. This is due to the fact that an *if* statement is compiled to *0x13* bytes, which is also added to the distance between relative jumps. To get *0xed* bytes, we use *j=0x1010101 7 times (0xb1*7=0xbd)* and *j++;j=i;j=i* (*0x20+0x8+0x8=0x30*). To generate *0xc35841* bytes (F2), we use *j=0x1010101 0xd3 times (0xb1*0xd3=0x1641)*, and fill the remaining *0xC34200* bytes by using *j++ 0x61A10 times (0x20*0x61A10=0xC34200)*.

The entire gadget generation process in Chrome took ≈ 1.3 seconds, in a VirtualBox Virtual Machine running Windows 10 (Intel Core i5-4690 CPU 3.50GHz).

Internet Explorer: As we have mentioned earlier, Internet Explorer, by default, comes with a 32-bit JIT compiler. Therefore, for gadget generation we chose gadgets that would be used in a 32-bit system. For simplicity

```

function popr8r9(r8,r9) {
  var i=0,j=0;
  if(r9) {
    /* F1: fillup 0xed Bytes */
    if (r8) {
      /* F2: fillup 0xc35841 Bytes */
    }
  }
}

```

Listing 4: JavaScript function *popr8r9*

```

function syscallIE() {
  var i=0;
  i=Math.random();
  ... /* 240 times in total */
  check_address();
  i=Math.random();
  ... /* 10 times in total */
  return i;
}

```

Listing 5: JavaScript function *syscallIE*

we used the set: `popa; int 0x80`, where `popa` sets the contents all x86 registers from the stack and `int 0x80` performs the system call.

The first part of the gadget emission process in IE is finding the right distance from the callee, i.e., a page that is `0xc3` bytes away from the callee. This part was done by a JavaScript code, which simply creates and compiles big functions (in our case 200 of them, $\approx 0x10\,000$ bytes each). After finding the correct page, we deallocated it and spammed the page with 16 specially crafted JavaScript functions, each of them covering `0x1000` bytes. For example, the JavaScript function used for emitting a system call (Listing 5) contains 250 `Math.random` calls (each of them compiling to `0xc` bytes). At the correct place between these calls, i.e., when the caller is at approximately the correct distance from `Math.random`, we inserted a call to `check_address` to verify the correctness of the gadget. In case the emitted call is not at the correct place, we deallocated the function and reallocated it again. Note that the reallocation is only needed for three-byte gadgets, where we also want to control the least significant byte. For two-byte gadgets (e.g., for `popa; ret`), we only need to call `check_address` to compute the address of the call instruction, for which we already know that is at the correct place (Listing 6).

In comparison to Chrome, gadget generation in IE is probabilistic and thus the time required for it also differs. There are two sources of the variance. First, generating the large functions to search for the correct third-byte distance from the callee; and second, compiling the gadget-emitting function in the found (correct) page, and

```

function poparetIE() {
  var i=0;
  i=Math.random();
  ... /* 232 times in total */
  check_address();
  i=Math.random();
  ... /* 28 times in total */
  return i;
}

```

Listing 6: JavaScript function *poparetIE*

Defense	Chrome	Firefox	IE
Const. Blinding	✓	✗	✓
NOP Insertion	✗	✗	✓
Code Size Limit	✗	✗	✓

Table 1: Current defenses in modern browsers

recompiling it until the correct gadget is emitted. In our experiments, we created 200 large functions and got the required third-byte distance for the first time in most of the cases. Compilation of these 200 functions took ≈ 4 seconds on a physical machine running Windows 10 (Intel Core i5-6200U CPU 2.3GHz). Each recompilation in the second step took 2-3 seconds. We ran the gadget generator in IE 10 times. Generating `popa; ret` and `int 0x80; ret` took on average 32 seconds, 11 and 47 seconds being the fastest and the slowest respectively.

4.4.2 Summary of Defenses and Vulnerabilities

We have shown that an attacker can encode arbitrary gadgets by triggering implicit constants with specially-crafted JavaScript code. Combining this with the ability to leak code pointers, an adversary can guess the addresses of the emitted gadgets without reading any code, thus making the attack possible even if code pages are non-readable.

Table 1 summarizes the defense techniques of modern browsers against code-reuse attacks in JIT-compiled code. Both IE and Chrome deploy constant blinding. Furthermore, IE uses NOP insertion as a fine-grained code randomization scheme, as also suggested in libbrando [17]. However, as Table 2 shows, none of the modern browsers sufficiently protect against the proposed attacks. Only Firefox “avoids” implicit constants by not using direct calls in baseline JIT compiler, but still exposes implicit constants in relative jumps.

¹Gadgets up to two bytes can be emitted.

Attack	Chrome	Firefox	IE
Relative Jumps	✗	✗	✗ ¹
Direct calls	✗	–	✗

Table 2: Browsers vulnerable to implicit constants

5 Defense

Seeing the threat of implicit constants, we now propose a technique to defend against it. We identify two steps that the attacker needs to take for using implicit constants as gadgets: (i) The attacker must be able to emit the required gadgets, and (ii) she must be able to acquire the necessary information (e.g., leak function pointers) to compute the addresses of the emitted gadgets.

One solution to tackle this problem would be to hide code pointers, e.g., by extending Readactor(++) to also cover the JIT-compiled code, as Crane *et al.* suggested. This would hinder the attacker from executing step (ii). However, this would still allow the attacker to emit arbitrary gadgets by leveraging the implicit constants (step (i)). Furthermore, the fact that the JIT compiler runs in the same process as the attacker makes it challenging to remove all possible code pointers that could, directly or indirectly, reveal the addresses of emitted gadgets. Therefore, we propose an orthogonal defense technique that forbids the attacker to emit the gadgets in the first place (i.e., step (i)). Our defense could be complemented with holistic code pointer hiding techniques to get additional security guarantees.

The main idea of our defense can be split in two parts: (i) We convert direct calls and jumps into indirect ones, such that their destination is taken from a register, and (ii) we use constant blinding to obfuscate the constants that are emitted by step (i) and may potentially contain attacker-controlled gadgets. For step (ii), we use the same cookie that is used by V8 to blind integer constants, and is generated anew before the compilation of each function. Note that the cookie is encoded in non-readable code and cannot be leaked. However, even if the attacker was able to leak the cookie, she could only guess the immediate values emitted in the current function, and any future function will have a different cookie value.

5.1 Removing Implicit Constants from V8

We integrated our defense into V8, Chrome’s JavaScript engine. We have chosen V8 due to its popularity and due to the fact that it is vulnerable to both our suggested attacks. Moreover, since V8 JIT-compiles JavaScript directly to the native code, it emits many checks (conditional jumps) and function calls (e.g., calls to inline

caches), which makes V8 a suitable candidate for our defense prototype evaluation. For our defense technique, we changed the functions of V8 that are responsible for emitting native code. In total, we modified ≈ 200 lines of code to account for all the cases of attacker controlled relative calls or jumps.

5.1.1 Conditional Jumps

To harden conditional jumps, we modified the native code that is emitted when JavaScript conditionals (such as `if`, `while`, `for`, `do-while`) are compiled. Our basic idea is to switch from relative to absolute jumps, and blind the resulting immediate values. To this end, we first add a padding (a sequence of NOP instructions) to each compiled conditional to reserve the space for later changes. For the hardened version of the conditional jump we need 19 bytes (instead of 6 bytes). We thus append 13 NOP instructions after the existing conditional jump. At the end of the compilation, when the constants of all jumps are calculated, we convert all relative jumps to absolute jumps, eliminating the need to fill a displacement with potential gadgets.

Figure 4 illustrates the steps of the aforementioned modifications. Figure 4(A) shows the compiled `if` statement in original V8. Figure 4(B) shows the same statement with the NOP padding. Finally, Figure 4(C) shows the assembly of the hardened `if` statement. In this final form, the condition of the original jump is inverted and the original long jump (having 4 byte jump distance) is replaced with the 1-byte short jump. Consequently, the new jump is taken if the original condition was false, i.e., the fall-through case. Otherwise, we convert the relative address into the absolute one, by adding it to the current instruction pointer (`rip`). This can be done with a single instruction in x64 (`lea r10, [rip+0xc380ca]`).

As this instruction will still emit the relative address as the displacement, we split it in two instructions. First, we add the current instruction pointer to the relative address AND-ed with a random key (`rip+0xc380ca&KEY`). In the second `lea` instruction, we add the sum to the relative address AND-ed with the inverted (bitwise not) random key, resulting in the desired offset (`rip+0xc380ca`). Note that we use obfuscation by AND-ing the constant with a random key instead of XOR-ing it, because $(A + B \oplus C) \oplus C$ does not equal to $A + B$, while $(A + B \wedge C) + B \wedge \neg C$ does. Moreover, this obfuscation scheme allows us to use `lea` instructions only, which has the advantage of not modifying any flags.

5.1.2 Direct Calls

We mitigate the implicit constants in direct calls by converting the direct calls into indirect ones. To this end, we

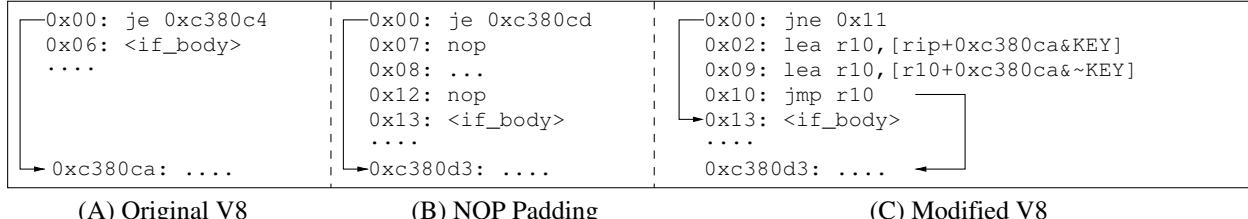


Figure 4: Steps of JIT hardening in V8

```

0x00: lea r10, [rip+ADDRESS&KEY ]
0x07: lea r10, [r10+ADDRESS&~KEY]
0x0e: call r10 ; calls 0xc380d3
0x11: ....

```

Figure 5: Hardening direct calls

distinguish whether the address of the callee is known at compile time (e.g., when calling built-in functions). If the callee’s address is known, we can move it to a scratch register (`r10`) and then execute an indirect call `mov r10, ADDRESS; call r10`. We thus emit the absolute address of the callee as the immediate value of the `mov` instruction, which is not under the control of the attacker and is thus safe—in contrast to the relative address.

If the address is unknown at compile time, we use a similar technique as we did for the conditional jumps, i.e., we convert the relative address into an absolute one (blinding the relative address during the conversion), store it in the scratch register, and then execute an indirect call, as shown in Figure 5.

5.2 Evaluation

To evaluate our defense technique we ran the V8 Benchmark Suite 7 on our modified V8. We performed each benchmark 100 times on both the modified and original V8s, and compared their corresponding averaged results. Table 3 illustrates the average scores that were returned by the benchmark suite, where a higher score indicates better performance. The modified V8 has an average overhead of less than 2%, and the worst overhead less than 3%. The observation that the overhead is negative for the *NavierStokes* benchmark can be explained by statistical variations across the different runs.

Additionally, we tested the modified V8 with microbenchmarks. To this end, we created two JavaScript functions (`ifs_true` and `ifs_false`), both of them containing 1,000,000 `if` statements. The condition of the `if` statement in `ifs_true` is always `true` (i.e., the `if` body is executed), while the condition of `ifs_false` is always `false`. This way the JIT-compiled functions will contain 1,000,000 conditional jump instructions modified by us,

each of them testing separate execution paths. Furthermore, evaluation of the expression in the `if` statements is done via a function call. Therefore, both of these functions generate 1,000,000 modified call instructions each and will thus incorporate the overhead caused by the function calls. Each run of the microbenchmark calls each of these functions 10 times. We ran the benchmark 1,000 times. We distinguish the first execution of these functions from the remaining nine, as the first execution is significantly slower due to the JIT-compiler modifying the generated intermediate functions to adjust them to the type information. Because the overhead was dominated mostly by the compiler, we did not see any overhead for the first function execution. For the remaining function executions we had 14.25% overhead in `ifs_false` and 9.81% for `ifs_true`.

Besides computational performance, our defense technique also causes a memory overhead due to added code. To measure this overhead, we compared the sizes of the functions compiled by the original and the modified versions of V8. To get the needed output from V8, we ran it with the `--print-code` flag, which outputs the disassembled code for each function after the compilation together with additional information about the compiled function including the size of the generated instructions. Running the benchmark suite with the aforementioned flag yielded that the total size of the instructions emitted by the original V8 was 1,123 kB, while the modified V8 emitted 1,411 kB, giving 287 kB of additional code, i.e., $\approx 26\%$ code size overhead. Given the significant size of the benchmark suite, and given that memory of nowadays x86/x64 systems are typically in the range of gigabytes, we think that hundreds of kB of additional code does not cause any bottlenecks on COTS systems.

6 Discussion

6.1 Defense Security Considerations

Our defense follows the general goal to remove unintended gadgets from constants in JIT-compiled code. We tailored our defense implementation towards protecting jump and call offsets. Other offsets may be usable to en-

Benchmark	Original	Modified	Overhead(%)
Richards	36,263	35,555	1.95
DeltaBlue	63,641	62,045	2.51
Crypto	33,366	32,725	1.92
RayTrace	77,198	75,488	2.21
EarleyBoyer	44,900	43,700	2.67
RegExp	6,525	6,414	1.71
Splay	21,095	20,479	2.92
NavierStokes	31,924	31,998	-0.23
Total	32,255	31,662	1.96

Table 3: Scores by the V8 benchmark (higher is faster)

code further gadgets. For example, relative addressing is frequently used in combination with the base pointer, such as when accessing parameters of a JavaScript function. As parameters are stored on the stack, they are accessed relative to the frame pointer (`ebp/rbp`). Each parameter access, after JIT-compilation, emits an assembly instruction, which contains the offset of the parameter from the frame pointer in its displacement field: `mov [ebp+0x0c],0x1`. The number of possible gadgets, in this technique, is restricted by (i) limited stack size (e.g., maximum $2^{16} - 1$ (`0xffff`) function parameters in Chrome) and (ii) stack alignment (4 or 8 bytes). In combination, this only allows generating gadgets whose opcodes are multiples of 4 (or 8) and are in the range between `0xc` and `0x40000`, and thus gives the attacker only limited capabilities. The stack size restrictions impose the same limitations on implicit gadgets encoded in relative accesses to function’s local variables.

While we think that the most important constants are blinded, we cannot exclude the existence of further ways to encode gadgets in assembly instructions. To eradicate all potential gadgets, one could prevent the JIT compiler from creating any potential gadgets (even in unaligned instructions). Most notably, G-Free [24] is a gadget-free compiler, which tries to generate gadget free binaries. However, G-Free requires multiple recompilations and code adjustments to reliably remove all possible gadgets. This will increase the runtime overhead for the JIT compilers, as the compilation time is included in their runtime.

6.2 Fine-Grained Code Randomization

An orthogonal approach to our defense would be to remove the attacker’s capability to find the gadget’s location (i.e., address). One way of doing so would be to hide code pointers, e.g., via trampolines, as suggested by Crane *et al.* [11]. If code pointers are not hidden, the attacker can read the return instruction pointer on the stack to get a pointer to the created gadget—which rep-

resents the current status in XnR implementations. This results in (i) getting access to the gadget, (ii) a possibility to verify the gadget at runtime, and (iii) the ability to retry in the case of a false result. By using similar techniques as we used against NOP insertion, the attacker can defeat fine-grained code randomizations such as the ones underlying Readactor [11]. Even though current XnR implementations do not hide code pointers in JIT-compiled code, XnR’s ideal implementation could also expand fully to the JIT-compiled code, e.g., by introducing trampolines. This, together with the fine-grained randomization schemes such as NOP insertion, would successfully protect against our attack. Note, however, that NOP insertion does not remove gadgets, but tries to reduce the chances of the attacker to guess their locations. In contrast, our proposed defense technique removes the gadgets, hence also removing the risk of the attacker doing a guesswork. Combining our technique with the extended XnR implementation would further improve the security guarantees, removing the chances of both emitting the gadgets and leaking the code layout information.

To guard against JIT-compiled gadgets, Wei *et al.* proposed to do several code modifications such as (i) securing immediate values via constant blinding, (ii) modifying internal fields of the instruction (e.g., registers being used), and (iii) randomizing the order of the parameters and local variables to randomize the offsets emitted by them [34]. However, this is not effective against the attacks proposed in Section 4, as the modifications do not secure the displacement fields emitted by relative calls/jumps. Finally, the code randomization proposed by Homescu *et al.* [17] that adds NOP instructions to randomize the code output from the JIT compiler remains ineffective if code pointers in JIT-compiled code are not hidden.

6.3 Attack Generality

A natural question is how the proposed attack generalizes, in particular to other operating systems or CPU architectures. We have evaluated the attacks against Chrome and Firefox running on Linux and IE on Windows. As we exploit properties of the JIT compilers to generate desired gadgets, the choice of the underlying operating system is arbitrary. The proposed attacks rely on the x86 system architecture (32- or 64-bit), though. In RISC architectures, such as ARM and MIPS, instruction lengths are fixed, and execution of unaligned instructions is forbidden by the hardware. However, the attacks may still apply to ARM, as an attacker could emit arbitrary two-byte values in the code if she can force the program to switch to 16-bit THUMB mode. Although this limits the attacker to using a single instruction, it still allows setting the register contents and diverting the control flow

at the same time, e.g., by using a `pop` instruction.

We implemented our defense in the 64-bit version of V8, taking advantage of the x64 architecture’s ability to directly read the instruction pointer (`rip`). This simplified the effort of converting relative addresses into absolute ones. Even though one can read the instruction pointer indirectly in 32-bit, e.g., by `call 0x0; pop eax`, such additional memory read instructions would increase the performance overhead. In addition, 32-bit features only eight general-purpose registers. While in x64 we could freely use a scratch register (`r10` for Chrome), in x86 we would likely need to save and restore the register. Similar defenses in x86-32 are thus possible, but come at an additional performance penalty. However, given that 64-bit systems are increasingly dominating the x86 market, we think that 64-bit solutions are most relevant.

7 Related Work

In the following, we will summarize existing code-reuse attacks and proposed defense mechanisms.

7.1 Code-Reuse Attacks: ROP / JIT-ROP

The most widespread defense against ROP is ASLR [32], which randomizes the base addresses of memory segments. Although it raises the bar, ASLR suffers from low entropy on 32-bit systems [30] and is not deployed in many libraries [28]. In addition, ASLR does not randomize within a memory segment, and thus leaves code at fixed offsets from the base address. Attackers can thus undermine ASLR by leaking a code pointer [18].

Researchers thus suggested fine-grained ASLR schemes that randomize code within segments. Fine-grained ASLR hides the exact code addresses from an attacker, even if a base pointer was leaked. For example, Pappas *et al.* [25] suggest diversifying code within basic blocks, such as by renaming and swapping registers, substituting instructions with semantically equivalent ones, or changing the order of register saving instructions. ASLP, proposed by Kil *et al.* [20], randomizes addresses of the functions as well as other data structures by statically rewriting an ELF executable. To increase the frequency of randomization, Wartell *et al.* propose STIR [33], which increases randomness by permuting basic blocks during program startup.

However, the invention of JIT-ROP undermined fine-grained ASLR schemes [31]. JIT-ROP assumes a memory disclosure vulnerability, which can be used by the attacker repeatedly. The attacker then follows the pointers to find executable memory, which she can read to find gadgets and build ROP chains on-the-fly.

Recently, Athanasakis *et al.* [1] proposed to extend JIT-ROP-like attacks by encoding gadgets in immediate

values of JIT-compiled code. Despite being limited to two-byte constant emission by IE, the authors managed to use aligned `ret` instructions, located at the end of each function, as the part of their gadget. Note that, in their attack, the authors were able to emit *complete* two-byte gadgets in IE. Therefore, this attack will be further limited against the 32-bit version of Chakra, which is a default JIT compiler, even for 64-bit IE. In addition, there are by now known defenses, such as constant blinding, that protect against explicit constants.

7.2 Hidden or Non-Readable Code

In reaction to JIT-ROP, researchers started to propose a great number of defensive schemes that try to hide code or function pointers. In Oxymoron, Backes *et al.* [3] aim to defend against JIT-ROP by hiding code pointers from direct calls. However, the attacker can still find indirect code pointers (e.g., return addresses on the stack or code pointers on the heap), and follow them to read the code. Davi *et al.* [13] thus proposed Isomeron, an improved defense. They keep two versions of the code at the same time, one original and another diversified using fine-grained ASLR. At each function call, they flip a coin to decide which version of the code to execute. This gives a 50% chance of success for each gadget in the chain, making it unlikely to guess correctly for long gadgets.

Gionta *et al.* [15] proposed HideM, which utilizes a split TLB to serve read and execute accesses separately, thus forbidding the attacker to read code pages. Apart from requiring hardware support, HideM also has a limitation that it does not protect function pointers, allowing the attacker to use them in code reuse attacks.

Backes *et al.* [2] and Crane *et al.* [11] proposed two independent defense techniques, XnR and Readactor, respectively, based on the same principle: making executable regions of the memory non-readable. XnR does this in software, marking executable pages non-present and checking the validity of the accesses in a custom page-fault handler. This leaves only a small window of (currently executing) readable code pages, significantly reducing the surface of gadgets an attacker can learn. Readactor uses Extended Page Tables (EPT), hardware-assisted virtualization support for modern CPUs. EPTs allow keeping all executable pages non-readable throughout the entire program execution. In addition, Readactor diversifies the *static* code of the program and hides addresses of the functions by introducing call/jump trampolines, making it impossible to guess the address of any existing code. While being effective against ROP attacks, Readactor left some pointers, such as function addresses in import tables and vtable pointers, intact, thus leaving the programs vulnerable against

function-wise code reuse attacks like return-to-libc [22] or COOP [27]. The fixes to these problems have been proposed by Crane *et al.* in their followup work Readactor++ [12]. We have demonstrated how an adversary can undermine these proposals if code pointer in JIT-compiled code are not hidden. As an orthogonal defense to hiding code pointer in JIT code, we proposed to eliminate implicit constants from JIT-compiled code to preserve XnR’s security guarantees.

Pereira *et al.* [26] designed a similar defense technique via a software-only approach for the ARM architecture. They propose Leakage-Resilient Layout Randomization (LR^2), which achieves non-readability of code in ARM by splitting the memory space in data and code pages and instrumenting load instructions to forbid code reading. Furthermore, LR^2 proposed to reduce the size overhead caused by trampolines by using a single trampoline for each callee and encoding the return address with secret per-function keys.

7.3 Defending JIT Against Attacks

Finally, we discuss research that aims to protect JIT compilers against exploitation. In JITDefender, Chen *et al.* [9] remove executable rights from the JIT-compiled code page until it is actually called by the compiler, and remove the rights when it is done executing. In this way they try to limit the time during which attackers can jump to JIT-sprayed shellcode. Although this was effective against some existing JIT-spraying attacks, JITDefender can be tricked by the attacker to keep the needed pages always executable, e.g., by keeping the executed code busy. Wu *et al.* [35] proposed RIM (Removing IMmediate), in which they rewrite instructions containing immediate values such that they cannot be used as a NOP sled. Later, Chen *et al.* [10] proposed to combine RIM and JITDefender, i.e., remove the executable rights from JIT-compiled code pages when not needed and also replace instructions containing immediate values.

In INSeRT, Wei *et al.* [34] propose fine-grained randomizations for JIT-compiled code. Their technique combines (i) removing immediate values via XORing them with random keys (i.e., constant blinding); (ii) register randomization; and (iii) displacement randomization (e.g., changing the order of parameters and local variables). Furthermore, INSeRT randomly inserts trapping instruction sequences, trying to catch attackers diverting the control flow. Still, its randomization neither affects call/jump displacements, nor would randomization without hiding code actually hinder our approach.

Most related to our attack are the defensive JIT randomization approaches proposed by Homescu *et al.* [17]. They propose librando, a library that uses NOP insertions to randomize the code offsets of JIT-compiled code. We

have demonstrated that even browsers leveraging NOP insertion (like IE) are susceptible to our proposed attack and thus proposed a non-probabilistic defense.

8 Conclusion

We have shown that commodity browsers do not protect against code reuse in attacker-generated, JIT-compiled code. Our novel attack challenges the assumption of XnR schemes in that we demonstrate that an attacker can create predictable ROP gadgets *without* the need to read them prior to use. To close this gap, we suggested to extend XnR schemes with our proposed countermeasure that eliminates all critical implicit constants in JIT-compiled code, effectively defending against our attack. Our defense evaluation shows that such practical defenses impose little performance overhead.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. Moreover, we are grateful for the guidance from our shepherd, Ben Livshits, during the process of finalizing the paper. We also want to thank Stefan Nürnberger, Dennis Andriesse, and David Pfaff for their comments during the writing process of the paper.

This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and for the BMBF project 13N13250.

References

- [1] ATHANASAKIS, M., ATHANASOPOULOS, E., POLYCHRONAKIS, M., PORTOKALIDIS, G., AND IOANNIDIS, S. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium* (February 2015).
- [2] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You Can Run but You Can’t Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS ’14, ACM, pp. 1342–1353.
- [3] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2014), SEC’14, USENIX Association, pp. 433–447.
- [4] BHATKAR, E., DUVARNEY, D. C., AND SEKAR, R. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium* (2003), pp. 105–120.
- [5] BHATKAR, S., AND SEKAR, R. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2008), DIMVA ’08, Springer-Verlag, pp. 1–22.

- [6] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), SSYM'05, USENIX Association, pp. 17–17.
- [7] BLAZAKIS, D. Interpreter Exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–9.
- [8] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 559–572.
- [9] CHEN, P., FANG, Y., MAO, B., AND XIE, L. JITDefender: A Defense against JIT Spraying Attacks. In *Future Challenges in Security and Privacy for Academia and Industry*, J. Camenisch, S. Fischer-Hbner, Y. Murayama, A. Portmann, and C. Rieder, Eds., vol. 354 of *IFIP Advances in Information and Communication Technology*. Springer Berlin Heidelberg, 2011, pp. 142–153.
- [10] CHEN, P., WU, R., AND MAO, B. JITSafe: a Framework against Just-in-time Spraying Attacks. *IET Information Security* 7, 4 (2013), 283–292.
- [11] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symposium on Security and Privacy (Oakland)* (May 2015).
- [12] CRANE, S., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., SUTTER, B. D., AND FRANZ, M. It's a TRAP: Table Randomization and Protection against Function Reuse Attacks. In *Proceedings of 22nd ACM Conference on Computer and Communications Security (CCS)* (2015).
- [13] DAVI, L., LIEBCHEN, C., SADEGHI, A.-R., SNOW, K. Z., AND MONROSE, F. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *22nd Annual Network & Distributed System Security Symposium (NDSS)* (Feb. 2015).
- [14] DAVI, L. V., DMITRIENKO, A., NÜRNBERGER, S., AND SADEGHI, A.-R. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 299–310.
- [15] GIONTA, J., ENCK, W., AND NING, P. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2015), CODASPY '15, ACM, pp. 325–336.
- [16] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. ILR: Where'D My Gadgets Go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 571–585.
- [17] HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Librando: Transparent Code Randomization for Just-in-time Compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 993–1004.
- [18] HUKU, A. Exploiting VLC. A Case Study on Jemalloc Heap Overflows. <http://www.phrack.org/issues/68/13.html>.
- [19] JACKSON, T., SALAMAT, B., HOMESCU, A., MANIVANNAN, K., WAGNER, G., GAL, A., BRUNTHALER, S., WIMMER, C., AND FRANZ, M. Compiler-Generated Software Diversity. In *Moving Target Defense*, S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, Eds., vol. 54 of *Advances in Information Security*. Springer New York, 2011, pp. 77–98.
- [20] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22Nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), ACSAC '06, IEEE Computer Society, pp. 339–348.
- [21] LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 276–291.
- [22] NERGAL. The Advanced Return-into-lib(c) Exploits. <http://phrack.org/issues/58/4.html>.
- [23] NOVARK, G., AND BERGER, E. D. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 573–584.
- [24] ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 49–58.
- [25] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 601–615.
- [26] PEREIRA, O., STANDAERT, F.-X., AND VIVEK, S. Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 96–108.
- [27] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (Oakland)* (May 2015).
- [28] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 25–25.
- [29] SHACHAM, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.
- [30] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 298–307.
- [31] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 574–588.
- [32] TEAM, P. Address Space Layout Randomization (ASLR). <http://pax.grsecurity.net/docs/aslr.txt>.

- [33] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 157–168.
- [34] WEI, T., WANG, T., DUAN, L., AND LUO, J. INSeRT: Protect Dynamic Code Generation against Spraying. In *Information Science and Technology (ICIST), 2011 International Conference on* (March 2011), pp. 323–328.
- [35] WU, R., CHEN, P., MAO, B., AND XIE, L. RIM: A Method to Defend from JIT Spraying Attack. In *Proceedings of the 2012 Seventh International Conference on Availability, Reliability and Security* (Washington, DC, USA, 2012), ARES '12, IEEE Computer Society, pp. 143–148.

zxcvbn: Low-Budget Password Strength Estimation

Daniel Lowe Wheeler
Dropbox Inc.

Abstract

For over 30 years, password requirements and feedback have largely remained a product of *LUDS*: counts of lower- and uppercase letters, **digits** and **symbols**. LUDS remains ubiquitous despite being a conclusively burdensome and ineffective security practice.

`zxcvbn` is an alternative password strength estimator that is small, fast, and crucially no harder than LUDS to adopt. Using leaked passwords, we compare its estimations to the best of four modern guessing attacks and show it to be accurate and conservative at low magnitudes, suitable for mitigating online attacks. We find 1.5 MB of compressed storage is sufficient to accurately estimate the best-known guessing attacks up to 10^5 guesses, or 10^4 and 10^3 guesses, respectively, given 245 kB and 29 kB. `zxcvbn` can be adopted with 4 lines of code and downloaded in seconds. It runs in milliseconds and works as-is on web, iOS and Android.

1 Introduction

Passwords remain a key component of most online authentication systems [32], but the quest to replace them [20] is an active research area with a long history of false starts and renewed enthusiasm (recently e.g., [33]). Whatever the future may hold for passwords, we argue that one of the most unusable and ineffective aspects of password authentication as encountered in 2016 truly does belong in the past: composition requirements and feedback derived from counts of lower- and uppercase letters, **digits** and **symbols** – *LUDS* for short.

LUDS requirements appear in many incarnations: some sites require digits, others require the presence of at least 3 character classes, some banish certain symbols, and most set varying length minimums and maximums [21, 29, 52]. It is also now commonplace to provide real-time password feedback in the form of visual strength bars and dynamic advice [50, 28]. As with pass-

word requirements, inconsistent LUDS calculations tend to lurk behind these feedback interfaces [25].

We review the history of LUDS in Section 2 as well as its usability problems and ineffectiveness at mitigating guessing attacks, facts that are also well known outside of the security community [23, 44]. But because it isn’t obvious what should go in place of LUDS, Section 3 presents a framework for evaluating alternatives such as `zxcvbn`. We argue that anything beyond a small client library with a simple interface is too costly for most would-be adopters, and that estimator accuracy is most important at low magnitudes and often not important past anywhere from 10^2 to 10^6 guesses depending on site-specific rate limiting capabilities.

The Dropbox tech blog presented an early version of `zxcvbn` in 2012 [55]. We’ve made several improvements since, and Section 4 presents the updated algorithm in detail. At its core, `zxcvbn` checks how common a password is according to several sources – common passwords according to three leaked password sets, common names and surnames according to census data, and common words in a frequency count of Wikipedia 1-grams. For example, if a password is the 55th most common entry in one of these ranked lists, `zxcvbn` estimates it as requiring 55 attempts to be guessed. Section 5.2.3 demonstrates that this simple minimum rank over several lists is responsible for most of `zxcvbn`’s accuracy.

Section 4 generalizes this idea in two ways. First, it incorporates other commonly used password patterns such as dates, sequences, repeats and keyboard patterns. Similar to its core, `zxcvbn` estimates the attempts needed to guess other pattern types by asking: if an attacker knows the pattern, how many attempts would they need to guess the instance? For example, the strength of the QWERTY pattern `zxcvfr` is estimated by counting how many other QWERTY patterns have up to six characters and one turn. Second, Section 4 generalizes beyond matching single patterns by introducing a search heuristic for multi-pattern sequences. That way, C0mpaq999

can be matched as the common password C0mpaq followed by the repeat pattern 999.

We simulate a professional guessing attack in Section 5 by running four attack models in parallel via Ur et al.’s Password Guessability Service [51, 8]. For each password in a test set sampled from real leaks, we take the minimum guess attempts needed over these four models as our conservative gold standard for strength. Section 5 measures accuracy by comparing strength estimations of each password to this gold standard. We investigate two other estimators in addition to zxcvbn: the estimator of KeePass Password Safe [5] (hereafter KeePass) as it is the only other non-LUDS estimator studied in [25],¹ and NIST entropy, an influential LUDS estimator reviewed in Section 2.

Our experiments are motivated by two intended uses cases: smarter password composition requirements and strength meters. Given a good estimator, we believe the best-possible strength requirement both in terms of usability and adopter control becomes a minimum check: does it take more than N attempts to guess the password? If not, tell the user their choice is too obvious and let them fix it however they want. For such a policy, an adopter needs confidence that their estimator won’t overestimate below their chosen N value. Similarly, smarter strength meters need to know over what range their estimator can be trusted.

Contributions:

- We demonstrate how choice of algorithm and data impacts accuracy of password strength estimation, and further demonstrate the benefit of matching patterns beyond dictionary lookup. We observe that KeePass and NIST substantially overestimate within the range of an online guessing attack and suggest a fix for KeePass.
- We show zxcvbn paired with 1.5MB of compressed data is sufficient to estimate the best-known guessing attacks with high accuracy up to 10^5 guesses. We find 245 kB is sufficient to estimate up to 10^4 guesses, and 29 kB up to 10^3 guesses.
- We present the internals of zxcvbn in detail, to serve immediately as a LUDS replacement for web and mobile developers, and in the future as a reference point that other estimators can measure against and attempt to beat.
- We present an evaluation framework for low-cost estimators to help future improvements balance security, usability, and ease of adoption.

¹We exclude Google’s estimator, a server-side estimator with no details available.

2 Background and Related Work

2.1 LUDS

LUDS has its roots at least as far back as 1985, tracing to the U.S. Defense Department’s influential *Password Management Guideline* [12] (nicknamed the Green Book) and NIST’s related *Password Usage of the Federal Information Processing Standards* series that same year [13]. The Green Book suggested evaluating password strength in terms of guessing space, modeled as $S = A^M$, where S is the maximum guess attempts needed to guess a password, M is the password’s length, and A is its alphabet size. For example, a length-8 password of random lowercase letters and digits would have a guessing space of $S = (26 + 10)^8$, and a passphrase of three random words from a 2000-word dictionary would have $S = 2000^3$. S is often expressed in bits as $M \cdot \log_2(A)$, a simple metric that is properly the *Shannon entropy* [47] when every password is assigned randomly in this way.

While this metric works well for machine-generated passwords, NIST’s related guideline applied similar reasoning to user-selected passwords at a time before much was known about human password habits. This erroneous randomness assumption persists across the Internet today. For example, consider our pseudocode summary of the metric NIST recommends (albeit with some disclaimers) in its most recent Special Publication 800-63-2 *Electronic Authentication Guideline* of August 2013 [22], commonly referred to as *NIST entropy* (hereafter NIST):

```

1: function NIST_ENTROPY( $p, dict$ )
2:    $e \leftarrow 4 + 2 \cdot p[2:8].len + 1.5 \cdot p[9:20].len + p[21:].len$ 
3:    $e \leftarrow e + 6$  if  $p$  contains upper and non-alpha
4:    $e \leftarrow e + 6$  if  $p.len < 20$  and  $p \notin dict$ 
5:   return  $e$ 
```

That is, NIST adds 4 bits for the first character in a password p , 2 bits each for characters 2 through 8, progressively fewer bits for each additional character, 6 more bits if both an uppercase and non-alphabetic character are used – so far, all LUDS. Up to now the 0 in Passw0rd would give it a higher entropy than QJqUbPpA. NIST recommends optionally adding up to 6 more bits if the password is under 20 characters and passes a dictionary check,² the idea being that longer passwords are likely to be multiword passphrases that don’t deserve a bonus. Even then, assuming Passw0rd fails the dictionary check and QJqUbPpA passes, these sample passwords would oddly have equal scores.

NIST entropy remains influential on password policy. Shay et al. [48] studied Carnegie Mellon University’s

²a non-LUDSian detail.

policy migration as part of joining the InCommon Federation and seeking its NIST-entropy-derived Silver Assurance certification [9], to give one notable example.

Whether used for feedback or for requirements, the goal of any LUDS formulation is ultimately to guide users towards less guessable passwords,³ and herein lies the first problem – it’s ineffective. Numerous studies confirm that people use types of characters in skewed distributions [27, 38, 48]: title case, all caps, digit suffixes, some characters more often than others within a class – to give only a small taste. Worse, the most commonly used patterns in passwords cannot be captured by character class counts, such as words, dates, and keyboard patterns. By taking tens of millions of leaked passwords and comparing NIST entropy to the guess order enumeration of a modern password cracker, Weir et al. [53] conclusively demonstrated that even with an added dictionary check and varied parameters, LUDS counts cannot be synthesized into a valid metric for password guessability. In a collaboration with Yahoo, Bonneau [19] found that a six-character minimum length policy exhibited almost no difference in online guessing resistance compared to no length requirement.

The second problem with LUDS is its high usability cost [34]. Any LUDS requirement beyond a low minimum length check necessarily disallows many strong passwords and places a burden on everyone, instead of a subgroup with a known risk of having their password guessed. Florêncio and Herley [29] studied the password policies of 75 American websites and concluded that policy stringency did not correlate with a heightened need for security, but rather with absence of competition and insulation from the consequences of poor usability.

This usability problem is compounded by policy inconsistency among sites. Wang and Wang [52] studied the password composition policies of 50 sites in 2015 (30 from mainland China, the rest mostly American) and found that no two sites enforced the same policy. Bonneau and Preibusch [21] found 142 unique policies in a 2010 survey of 150 high-traffic sites. As a result of these inconsistent policies, people often have to jump through unique hoops upon selecting new passwords [23].

Password feedback is similarly inconsistent across the Internet. Carnavalet and Mannan [25] investigated the registration flows of 18 prominent services spanning multiple countries, platforms, and product domains, and with three exceptions (Google, Dropbox, and KeePass), they found simple but widely inconsistent LUDS-based calculations powering visual feedback, sometimes combined with dictionary checks.

³Proximate goals might include compliance or perception of security, both of which still derive from a notion of guessing resistance in most instances.

2.2 Password Guessing

In their seminal 1979 publication, Morris and Thompson [43] conducted one of the first studies of password habits and detailed the early UNIX history of co-evolving password attacks and mitigations. The decades that followed have seen immense development in password guessing efficiency.

Our gold standard for measuring a password’s *strength* is the minimum attempts needed to guess it over four modern guessing attacks, obtained by running the `min_auto` configuration of Ur et al.’s Password Guessability Service [8] (hereafter PGS), demonstrated in [51] to be a conservative estimate of an experienced and well-resourced professional. We run two cutting-edge attacks from the literature, consisting of a PCFG model [54] with Komanduri’s improvements [36] and a Smoothed Order-5 Markov model [39]. Because our gold standard should be a safe lower bound not just over the theoretical best attacks, but also the best-productized attacks in common use by professionals, we further run Hashcat [2] and John the Ripper [3] mangled dictionary models with carefully tuned rule sets.

Throughout this paper, we will differentiate between *online guessing*, where an attacker attempts guesses through a public interface, and *offline guessing*, where, following a theft of password hashes, an attacker makes guesses at a much higher rate on their own hardware. We recommend [30] for a more detailed introduction to guessing attacks.

2.3 Guessing Resistance

We focus on guessing resistance techniques that influence usability, as opposed to developments in cryptography, abuse detection, and other service-side precautions.

The idea of a *proactive password checker*, a program that offers feedback and enforces policy at composition time, traces to the late 80s and early 90s with pioneering work by Nagle [45], Klein [35], Bishop [17] and Spafford [49]. Eight days after the Morris worm, which spread in part by guessing weak passwords, Nagle presented his *Obvious Password Detector* program that rejected any password that didn’t have a sufficient number of uncommon triplets of characters. Klein focused on dictionary checks with various transformations, such as reversed token lookup, common character substitutions, and stemming, but also recommended LUDS rules including the rejection of all-digit passwords. In his `pwcheck` program, Bishop introduced a concise language for system administrators to formulate composition rules in terms of dictionary lookups, regular expression matching, and relations to personal information. This language is also one of the first to allow customized

user feedback, a precursor to today’s ubiquitous password strength meters. None of these early rule-based systems estimate password strength, making it hard to correctly balance usability and answer whether a password is sufficiently unguessable.

Spafford and others proposed space-efficient dictionary lookup techniques using Bloom filters [49, 40] and decision trees [15, 18]. These approaches similarly do not directly estimate guessing resistance or compare their output to modern guessing attacks, providing instead a binary pass/fail. Yan [56] highlights the need to catch patterns beyond dictionary lookups, something we find modest supporting evidence for in Section 5.2.3.

Castelluccia et al. [24] propose maintaining a production database of character *n*-gram counts in order to model a password’s guessability with an adaptive Markov model. Schechter et al. [46] outline a *count-min sketch* datastructure to allow all passwords that aren’t already too popular among other users. Both of these proposals have the advantage of modeling a service’s unique password distribution. Both aggregate information that, if stolen, aid offline cracking attacks, and both include noise mitigations to reduce that threat. We argue in Section 3 that their respective requirements to maintain and secure custom production infrastructure at scale is too costly for most would-be adopters.

Dell’Amico and Filippone [26] detail a Monte Carlo sampling method that converts a password’s probability as computed by any generative model into an estimate of a cracker’s guessing order when running that model. Given that some of today’s best guessing techniques are built on probabilistic models [39, 54, 36], the benefit of this approach is fast and accurate estimation of guessing order, even up to as high as 2^{80} (10^{24}) guesses. But while the conversion step itself is time- and space-efficient, we haven’t encountered investigations in the literature that limit the size of the underlying probability model to something that would fit inside a mobile app or browser script. Comparing space-constrained probabilistic estimators to today’s best guessing attacks (or perhaps a minimum over several parallel attacks as we do) would be valuable future work. Melicher et al.’s concurrent and independent research on lean estimation with Recurrent Neural Networks is quite promising [42].

Turning to open-source industry contributions, `zxcvbn` and KeePass [5] were originally designed for password strength feedback, but we consider them here for policy enforcement as well. Industry adoption of `zxcvbn` is growing, currently deployed by Dropbox, Stripe, Wordpress, Coinbase, and others. KeePass (reviewed in [25]) matches several common patterns including a dictionary lookup with common transformations, then applies an *optimal static entropy encoder* documented in their help center [4] to search for the

simplest set of candidate matches covering a password. We extracted KeePass into a stand-alone command-line utility such that we could compare it against realistic guessing attacks in Section 5.

Telepathwords [37] offers some of the best real-time password feedback currently available. It employs a client-server architecture that is hosted as a stand-alone site, and does not output a guess attempt estimate or equivalent, so we do not evaluate it as a candidate LUDS alternative.

Ur et al. [50] and Egelman et al. [28] studied the effect of strength meters on password composition behavior. The consensus is that users do follow the advice of these meters for accounts of some importance; however, both studies employed LUDS metrics to power their meters as is common in the wild, conditionally with an added dictionary check in the case of [50]. Our aim is to provide strength meters with more accurate underlying estimation.

3 Evaluation Framework

While the problems of LUDS are well understood, it isn’t obvious what should go in its place. We motivate some of the important dimensions and reference two LUDS methods for comparison: NIST as well 3class8 – the easy-to-adopt requirement that a password contain 8 or more characters of at least 3 types.

It should be no harder than LUDS to adopt

In the wider scheme of password authentication, composition policy and feedback are small details. Bonneau and Preibusch [21] demonstrated that the big details – cryptography and rate-limiting, for example – are commonly lacking throughout the Internet with little economic incentive to do better. To then have a starting chance of widespread adoption, a LUDS alternative cannot be harder than LUDS to adopt. We believe this realistically eliminates alternatives that require hosting and scaling special infrastructure from mainstream consideration, whereas small client libraries with simple interfaces are more viable. To give an example, the following is working web integration code for a policy that disallows passwords guessable in under 500 attempts according to `zxcvbn`:

```
var zxcvbn = require('zxcvbn');
var meets_policy = function(password) {
    return zxcvbn(password).guesses > 500;
};
```

This sample assumes a CommonJS module interface. For comparison, Appendix A lists our implementation of 3class8 back-to-back with two other `zxcvbn` integration options.

It should only burden at-risk users

Users face many threats in addition to guessing attacks, including phishing, keylogging, and credential reuse attacks. Because guessing attacks often rank low on a user’s list of worries, short and memorable password choices are often driven by rational cost-benefit analysis as opposed to ignorance [31]. To encourage less guessable choices, a LUDS alternative must accept this reality by imposing as few rules as possible and burdening only those facing a known guessing risk. As examples, words should be recognized and weighted instead of, as with space-efficient dictionary lookups, rejected. All-digit and all-lowercase passwords – respectively the most common password styles in mainland China and the U.S. according to a comprehensive study [38] – should similarly be weighted instead of rejected via blanket rules. `3class8` is a prime offender in this category.

While underestimation harms usability, overestimation is arguably worse given an estimator’s primary goal of mitigating guessing attacks. In Section 5 we measure accuracy and overestimation as separate quantities in our comparison of alternative estimators. We find KeePass and NIST tend to overestimate at lower magnitudes.

It should estimate guessing order

The security community’s consensus definition of a password’s strength is *the number of attempts that an attacker would need in order to guess it* [26]. Strength estimators should thus estimate this guessing order directly, versus an entropy, percentage, score, or binary pass/fail. This detail provides adopters with an intuitive language for modeling guessing threats and balancing usability. An alternative should further measure its estimations against real guessing attacks and communicate its accuracy bounds. For example, given enough samples, [26] is accurate up to the highest feasible guessing ranges, whereas with 1.5MB of data, `zxcvbn` is only highly accurate up to 10^5 guesses.

It should be accurate at low magnitudes

Online guessing attacks are a threat that password authentication systems must defend against continually. While rate limiting and abuse detection can help, passwords guessable in, say, 10 to 100 guesses could remain vulnerable to capable adversaries over time. As we show in Section 5.2, NIST greatly overestimates password strength at low magnitudes. Similarly, `3class8` permits obvious choices such as `Password1!`. A strict improvement over LUDS in terms of security, then, is to improve accuracy at low guessing magnitudes.

Past an online guessing threshold, the benefit of accurate estimation becomes more situation-dependent. Referencing the *online-offline chasm* of [30], the added security benefit from encouraging or requiring stronger passwords might only start to appear after many orders of

magnitude past an online guessing cutoff, indicating a substantial usability-security trade-off that often won’t be justified. While we focus on online attack mitigation, key stretching techniques such as Argon2 [16] can further render offline attacks unfeasible at higher guessing magnitudes.

For the remainder of this paper, we will use 10^6 guesses as our approximate online attack cutoff, citing the back-of-the-envelope upper limit estimation in [30] for a depth-first online attack (thus also bounding an online trawling attack). By studying leaked password distributions, [30] also points out that an attacker guessing in optimal order would face a reduction in success rate by approximately 5 orders of magnitude upon reaching 10^6 guesses.

While we use 10^6 as an online cutoff for safe and simple analysis in Section 5, we recognize that an upper bound on online guessing is highly dependent on site-specific capabilities, and that some sites will be able to stop an online attack starting at only a few guesses. This motivates our next item:

It should have an adjustable size

An estimator’s accuracy is greatly dependent on the data it has available. Adopters should be given control over this size / accuracy trade-off. Some might want to bundle an estimator inside their app, selecting a smaller size. We expect most will want to asynchronously download an estimator in the background on demand, given that password creation only happens once per user and typically isn’t the first step in a registration flow. Current bandwidth averages should factor into this discussion: a South Korean product might tolerate a gzipped-5MB estimator (downloadable in 2 seconds at 20.5Mbps national average in Q3 2015 [14]⁴), whereas 1.5MB is a reasonable global upper bound in 2016 (2.3 seconds at 5.1Mbps Q3 2015 global default). Need should also factor in: a site that is comfortable in its rate-limiting might only need accurate estimation up to 10^2 guesses.

4 Algorithm

We now present the internals of `zxcvbn` in detail. Sections 4.2 and 4.3 explain how common token lookup and pattern matching are combined into a single guessing model. Section 4.4 is primarily about speed, providing a fast algorithm for finding the simplest set of matches covering a password. We start with a high-level sketch.

4.1 Conceptual Outline

`zxcvbn` is non-probabilistic, instead heuristically estimating a guessing attack directly. It models passwords

⁴We cite figures from Akamai’s State of the Internet series.

as consisting of one or more concatenated patterns. The 2012 version of `zxcvbn` assumes the guesser knows the pattern structure of the password it is guessing, with bounds on how many guesses it needs per pattern. For example, if a password consists of two top-100 common words, it models an attacker who makes guesses as concatenations of two words from a 100-word dictionary, calculating 100^2 as its worst-case guess attempt estimate.

To help prevent overly complex matching, `zxcvbn` now loosens the 2012 assumption by instead assuming the attacker knows the patterns that make up a password, but not necessarily how many or in which order. To illustrate the difference, compare the respective 2012 and 2016 analyses of `jessiah03`:

```
jess(name) i(word) ah0(surname) 3(bruteforce)
jessia(name) h03(bruteforce)
```

The 2012 version has no bias against long pattern sequences, matching `i` and `jess` as common words. (`jessia` is in the common surname dictionary at about rank 3.5k, `jess` is at about rank 440, and `jessiah` is in neither dictionary.) To give another example, the random string `3vMs3o7B7eTo` is now matched as a single brute-force region by `zxcvbn`, but as a 5-pattern sequence by the 2012 version, including `7eT` as a 133ted “let.”

To formalize this difference in behavior, at a high level, both versions consist of three phases: match, estimate and search. Given a plaintext password input, the pattern matching phase finds a set S of overlapping matches. For example, given `lenovo1111` as input, this phase might return `lenovo` (password token), `eno` (English “one” backwards), `no` (English), `no` (English “on” backwards), `1111` (repeat pattern), and `1111` (Date pattern, 1/1/2011). Next, the estimation phase assigns a guess attempt estimation to each match independently. If `lenovo` is the 11007th most common password in one of our password dictionaries, it’ll be assigned 11007, because an attacker iterating through that dictionary by order of popularity would need that many guesses before reaching it. The final phase is to search for the sequence of non-overlapping adjacent matches S drawn from \mathcal{S} such that S fully covers the password and minimizes a total guess attempt figure. In this example, the search step would return [`lenovo` (token), `1111` (repeat)], discarding the date pattern which covers the same substring but requires more guesses than the repeat. `zxcvbn`’s formalized assumption about what an attacker knows is represented by the following search heuristic:

$$\underset{S \subseteq \mathcal{S}}{\operatorname{argmin}} D^{|S|-1} + |S|! \prod_{m \in S} m.\text{guesses} \quad (1)$$

$|S|$ is the length of the sequence S , and D is a constant. The intuition is as follows: if an attacker knows the pattern sequence with bounds on how many guesses needed

for each pattern, the Π term measures how many guesses they would need to make in the worst case. This Π term, by itself, is the heuristic employed by the 2012 version. With the added $|S|!$ term, the guesser now knows the number of patterns in the sequence but not the order. For example, if the password contains a common word c , uncommon word u , and a date d , there are $3!$ possible orderings to try: `cud`, `ucd`, etc.

The $D^{|S|-1}$ term attempts to model a guesser who additionally doesn’t know the length of the pattern sequence. Before attempting length- $|S|$ sequences, `zxcvbn` assumes that a guesser attempts lower-length pattern sequences first with a minimum of D guesses per pattern, trying a total of $\sum_{l=1}^{|S|-1} D^l \approx D^{|S|-1}$ guesses for sufficiently large D . For example, if a password consists of the 20th most common password token t with a digit d at the end – a length-2 pattern sequence – and the attacker knows the $D = 10000$ most common passwords, and further, td is not in that top-10000 list (otherwise it would have been matched as a single token), the D^1 term models an attacker who iterates through those 10000 top guesses first before moving on to two-pattern guessing. While an attacker might make as few as 10 guesses for a single-digit pattern or as many as tens of millions of guesses iterating through a common password dictionary, we’ve found $D = 1000$ to $D = 10000$ to work well in practice and adopt the latter figure for `zxcvbn`.

In practical terms, the additive D penalty and multiplicative $|S|!$ penalty address overly complex matching in different ways. When two pattern sequences of differing length have near-equal Π terms, the $|S|!$ factor biases towards the shorter sequence. The D term biases against long sequences with an overall low Π term.

4.2 Matching

The matching phase finds the following patterns:

pattern	examples
<code>token</code>	<code>logitech</code> <code>10giT3CH</code> <code>ain't</code> <code>parliamentarian</code> <code>1232323q</code>
<code>reversed</code>	<code>Drowssap</code>
<code>sequence</code>	<code>123</code> <code>2468</code> <code>jklm</code> <code>ywusq</code>
<code>repeat</code>	<code>zzz</code> <code>ababab</code> <code>10giT3CH10giT3CH</code>
<code>keyboard</code>	<code>qwertyuiop</code> <code>qAzxcde3</code> <code>diueoa</code>
<code>date</code>	<code>7/8/1947</code> <code>8.7.47</code> <code>781947</code> <code>4778</code> <code>7-21-2011</code> <code>72111</code> <code>11.7.21</code>
<code>bruteforce</code>	<code>x\$JQhMzt</code>

The token matcher lowerscases an input *password* and checks membership for each substring in each frequency-ranked dictionary. Additionally, it attempts

each possible l33t substitution according to a table. An input @BA1one is first lowercased to @ba1one. If the l33t table maps @ to a and 1 to either i or l, it tries two additional matches by subbing [$\text{@}\rightarrow\text{a}$, $\text{l}\rightarrow\text{i}$] and [$\text{@}\rightarrow\text{a}$, $\text{l}\rightarrow\text{l}$], finding abalone with the second substitution.

Taking a cue from KeePass, sequence matching in zxvcvbn looks for sequences where each character is a fixed Unicode codepoint distance from the last. This has two advantages over the hardcoded sequences of the 2012 version. It allows skipping, as in 7531, and it recognizes sequences beyond the Roman alphabet and Arabic numerals, such as Cyrillic and Greek sequences. Unicode codepoint order doesn't always map directly to human-recognizable sequences; this method imperfectly matches Japanese kana sequences as one example.

The repeat matcher searches for repeated blocks of one or more characters, a rewrite of the 2012 equivalent, which only matched single-character repeats. It tries both greedy $(.+)\backslash 1+$ and lazy $(.+)?\backslash 1+$ regular expressions in search of repeated regions spanning the most characters. For example, greedy beats lazy for aabaab, recognizing (aab) repeated over the full string vs (a) repeated over aa, whereas lazy beats greedy for aaaaa, matching (a) spanning 5 characters vs (aa) spanning 4. The repeat matcher runs a match-estimate-search recursively on its winning repeated unit, such that, for example, repeated words and dates are identified.

The keyboard matcher runs through *password* linearly, looking for chains of adjacent keys according to each of its keyboard adjacency graphs. These graphs are represented as a mapping between each key to a clockwise positional list of its neighbors. The matcher counts chain length, number of turns, and number of shifted characters. On QWERTY, zxvcvfr\$321 would have length 10, 2 turns, and 2 shifted characters. QWERTY, DVORAK, and Windows and Mac keypad layouts are included by default. Additional layouts can be prepackaged or dynamically added.

Date matching considers digit regions of 4 to 8 characters, checks a table to find possible splits, and attempts a day-month-year mapping for each split such that the year is two or four digits, the year isn't in the middle, the month is between 1 and 12 inclusive, and the day is between 1 and 31 inclusive. For example, a six-digit sequence 201689 could be broken into 2016-8-9, 20-16-89, or 2-0-1689. The second candidate would be discarded given no possible month assignment, and the third discarded because 0 is an invalid day and month. Given multiple valid splits, the choice with year closest to a reference year of 2016 wins. Two-digit years are matched as 20th- or 21st-century years, depending on whichever is closer to 2016. For ease of portability, date matching does not filter improper Gregorian dates; for example, it allows Feb. 29th on a non-leap year.

4.3 Estimation

Next, a guess attempt estimate *guesses* is determined for each match $m \in \mathcal{S}$. The guiding heuristic is to ask: if an attacker knows the pattern, how many guesses might they need to guess the instance? Green Book-style guessing space calculations then follow, but for patterns instead of random strings, where a guesser attempts simpler or more likely patterns first.

For tokens, we use the frequency rank as the estimate, because an attacker guessing tokens in order of popularity would need at least that many attempts. A reversed token gets a doubled estimate, because the attacker would then need to try two guesses (normal and reversed) for each token. A conservative factor of 2 is also added for an obvious use of uppercase letters: first-character, last-character, and all caps. The capitalization factor is otherwise estimated as

$$\frac{1}{2} \sum_{i=1}^{\min(U,L)} \binom{U+L}{i} \quad (2)$$

where U and L are the number of uppercase and lowercase letters in the token. For example, to guess paSsw0rd, an attacker would need to try a guessing space of 8 different single-character capitalizations plus 28 different two-character capitalizations. The $1/2$ term converts the total guessing space into an average attempts needed, assuming that each capitalization scheme is equally likely – this detail could be improved by better modeling observed distributions of capitalization patterns in leaked password corpora. The $\min()$ term flips the lowercasing game into an uppercasing game when there are more upper- than lowercase letters, yielding 8 for PAsWORD.

Guesses for keyboard patterns are estimated as:

$$\frac{1}{2} \sum_{i=1}^L \sum_{j=1}^{\min(T,i-i)} \binom{i-1}{j-1} SD^j \quad (3)$$

where L is the length of the pattern, T is the number of terms, D is the average number of neighbors per key (a tilde has one neighbor on QWERTY, the ‘a’ key has four) and S is the number of keys on the keyboard. For T turns throughout a length L keyboard pattern, we assume a guesser attempts lower-length, lower-turn patterns first, starting at length 2. The binomial term counts the different configuration of turning points for a length- i pattern with j turns, with -1 added to each term because the first turn is defined to occur on the first character. The $\min()$ term avoids considering more turns than possible on a lower-length pattern. The sequence might have started on any of S keys and each turn could have gone any of D ways on average, hence the $S \cdot D^j$. Equation 3 estimates about 10^3 guesses for kjhgfdsa on QWERTY and 10^6

guesses for `kjhgt543`. Shifted keys in the pattern add a factor according to expression 2, where L and U become shifted and unshifted counts.

Repeat match objects consist of a base repeated n times, where a recursive match-estimate-search step previously assigned a number of guesses g to the base. Repeat guess attempts are then estimated as $g \cdot n$. For example, `nownownow` is estimated as requiring 126 guesses: now is at rank 42 in the Wiktionary set, times 3.

Sequences are scored according to $s \cdot n \cdot |d|$, where s is the number of possible starting characters, n is the length, and d is the codepoint delta (e.g., -2 in 9753). s is set to a low constant 4 for obvious first choices like 1 and Z, set to 10 for other digits, or otherwise 26, an admittedly Roman-centric default that could be improved.

For dates, we assume guessers start at 2016 and guess progressively earlier or later dates, yielding a ballpark of $365 \cdot |2016 - \text{year}|$

Finally, bruteforce matches of length l are assigned a constant $C = 10$ guesses per character, yielding a total estimate of C^l . The 2012 version performs a guessing space calculation, treating bruteforce regions as random, and determines a cardinality C that adds 26 if any lowercase letters are present, 26 if uppercase, 10 if digits, and 33 for one or more symbols. This dramatically overestimates the common case, for example a token that isn't in the dictionary. The 2012 version scores `Teiubesc` (Romanian for "I love you") as $(26 + 26)^8 \approx 10^{14}$, whereas `zxcvbn` now estimates it 6 orders of magnitude lower at 10^8 . (Thanks to the addition of RockYou'09 data, it also matches it as a common password at rank 10^4).

4.4 Search

Given a string `password` and a corresponding set of overlapping matches \mathcal{S} , the last step is to search for the non-overlapping adjacent match sequence S that covers `password` and minimizes expression (1). We outline a dynamic programming algorithm that efficiently accomplishes this task. The idea is to iteratively find the optimal length- l sequence of matches covering each length- k character prefix of `password`. It relies on the following initial state:

```

1:  $n \leftarrow \text{password.length}$ 
2:  $\mathcal{B}_{opt} \leftarrow [] \times n$ 
3:  $\Pi_{opt} \leftarrow [] \times n$ 
4:  $l_{opt} \leftarrow 0$ 
5:  $g_{opt} \leftarrow \text{null}$ 

```

\mathcal{B}_{opt} is a backpointer table, where $\mathcal{B}_{opt}[k][l]$ holds the ending match in the current optimal length- l match sequence covering the length- k prefix of `password`. $\Pi_{opt}[k][l]$ correspondingly holds the product term in ex-

pression (1) for that sequence. When the algorithm terminates, g_{opt} holds the optimum guesses figure and l_{opt} holds the length of the corresponding optimal sequence. Note that if no length- l sequence exists such that it scores lower than every alternative sequence with fewer matches covering the same k -prefix, then $l \notin \mathcal{B}_{opt}[k]$.

Each match object m has a guess value $m.guesses$ and covers a substring of `password` at indices $m.i$ and $m.j$, inclusive. The search considers one character of `password` at a time, at position k , and for each match m ending at k , evaluates whether adding m to any length- l optimal sequence ending just before m (at $m.i - 1$) leads to a new candidate for the optimal match sequence covering the prefix up to k :

```

1: function SEARCH( $n, \mathcal{S}$ )
2:   for  $k \in 0$  to  $n - 1$ 
3:      $g_{opt} \leftarrow \infty$ 
4:     for  $m \in \mathcal{S}$  when  $m.j = k$ 
5:       if  $m.i > 0$ 
6:         UPDATE( $m, l + 1$ ) for  $l \in \mathcal{B}_{opt}[m.i - 1]$ 
7:       else
8:         UPDATE( $m, 1$ )
9:       BF_UPDATE( $k$ )
10:      return UNWIND( $n$ )

```

Instead of including a bruteforce match object in \mathcal{S} for every $O(n^2)$ substring in `password`, bruteforce matches are considered incrementally by BF_UPDATE:

```

1: function BF_UPDATE( $k$ )
2:    $m \leftarrow \text{bruteforce from } 0 \text{ to } k$ 
3:   UPDATE( $m, 1$ )
4:   for  $l \in \mathcal{B}_{opt}[k - 1]$ 
5:     if  $\mathcal{B}_{opt}[k - 1][l]$  is bruteforce
6:        $m \leftarrow \text{bruteforce from } \mathcal{B}_{opt}[k - 1][l].i \text{ to } k$ 
7:       UPDATE( $m, l$ )
8:     else
9:        $m \leftarrow \text{bruteforce from } k \text{ to } k$ 
10:      UPDATE( $m, l + 1$ )

```

That is, at each index k , there are only three cases where a bruteforce match might end an optimal sequence: it might span the entire k -prefix, forming a length-1 sequence, it might extend an optimal bruteforce match ending at $k - 1$, or it might start as a new single-character match at k . Note that given the possibility of expansion, it is always better to expand by one character than to append a new bruteforce match, because either choice would contribute equally to the Π term, but the latter would increment l .

The UPDATE helper computes expression (1) and updates state if a new minimum is found. Thanks to the Π_{opt} table, it does so without looping:

```

1: function UPDATE( $m, l$ )
2:    $\Pi \leftarrow m.guesses$ 
3:   if  $l > 1$ 
4:      $\Pi \leftarrow \Pi \times \Pi_{opt}[m.i - 1][l - 1]$ 
5:    $g \leftarrow D^{l-1} + l! \times \Pi$ 
6:   if  $g < g_{opt}$ 
7:      $g_{opt} \leftarrow g$ 
8:      $l_{opt} \leftarrow l$ 
9:    $\Pi_{opt}[k][l] \leftarrow \Pi$ 
10:   $\mathcal{B}_{opt}[k][l] \leftarrow m$ 

```

At the end, UNWIND steps through the backpointers to form the final optimal sequence:

```

1: function UNWIND( $n$ )
2:    $S \leftarrow []$ 
3:    $l \leftarrow l_{opt}$ 
4:    $k \leftarrow n - 1$ 
5:   while  $k \geq 0$ 
6:      $m \leftarrow \mathcal{B}_{opt}[k][l]$ 
7:      $S.prepend(m)$ 
8:      $k \leftarrow m.i - 1$ 
9:      $l \leftarrow l - 1$ 
10:    assert  $l = 0$ 
11:    return  $S$ 

```

Each match $m \in \mathcal{S}$ is considered only once during the search, yielding a runtime of $O(l_{max} \cdot (n + |\mathcal{S}|))$, where l_{max} is the maximum value of l_{opt} over each k iteration. In practice, l_{max} rarely exceeds 5, and this method rapidly terminates even for passwords of hundreds of characters and thousands of matches.

4.5 Deployment

`zxcvbn` is written in CoffeeScript and compiled via an npm build flow into both a server-side CommonJS module and a minified browser script. The ranked token lists take up most of the total library size: each is represented as a sorted comma-separated list of tokens which then get converted into an object, mapping tokens to their ranks. The browser script is minified via UglifyJS2 with instructions on how to serve as gzipped data.

`zxcvbn` works as-is on most browsers and javascript server frameworks. Because iOS and Android both ship with javascript interpreters, `zxcvbn` can easily interface with most mobile apps as well. JSContext on iOS7+ or UIWebView for legacy support both work well. Running javascript with or without a web view works similarly on Android.

Dropbox uses `zxcvbn` for feedback and has never enforced composition requirements other than a 6-character minimum. For those implementing requirements, we suggest a client-side soft enforcement for simplicity,

such as a submit button that is disabled until the requirement is met. Because different versions and ports give slightly different estimates, we suggest those needing server-side validation either skip client-side validation or make sure to use the exact same build across their server and various clients. `zxcvbn` ports exist for Java, Objective-C, Python, Go, Ruby, PHP, and more.

4.6 Limitations

`zxcvbn` doesn't model interdependencies between patterns, such as common phrases and other collocations. However, its ranked password dictionaries include many phrases as single tokens, such as `opensesame`. It only matches common word transformations that are easy to implement given limited space; it doesn't match words with deleted letters and other misspellings, for example. Unmatched regions are treated equally based on length; the English-sounding made-up word `novanoid` gets the same estimate as a length-8 random string, and unmatched digits and symbols are treated equally even though some are more common than others.

5 Experiments

We investigate how choice of algorithm and dataset impacts the estimation accuracy of a realistic guessing attack. We also show the impact of matching patterns beyond token lookup. Our experiments employ a test set of 15k passwords from the RockYou'09 leak [7]. Appendix C includes the same analysis on a 15k sample from the Yahoo'12 leak [11]. We close the Section with runtime benchmarks for `zxcvbn`.

5.1 Methodology

Algorithms and Data

The algorithms we selected for our experiment – NIST, KeePass, and `zxcvbn` – estimate guess attempts or equivalent (excludes [49, 17]) and can operate without a backing server (excludes [46, 24]).

For our password strength gold standard, as introduced in Section 2.2, we ran the `min_auto` configuration of PGS [8] with the same training data found to be most effective in [51]. The PGS training data (roughly 21M unique tokens) consists of the RockYou'09 password leak (minus a randomly sampled 15k test set), Yahoo'12 leak (minus a similar 15k test set), MySpace'06 leak, 1-grams from the Google Web Corpus, and two English dictionaries. To make brute force guessing attacks infeasible, the 15k test sets are sampled from the subset of passwords that contain at least 8 characters. Of the four attacks, we ran the Markov attack up through 10^{10} guesses, John the Ripper and Hashcat up through 10^{13} ,

and PCFG up to 10^{14} . Detailed specifics can be found in [51].

While our test data is distinct from our training data, it is by design that both include samples from the same RockYou’09 distribution; our aim is to simulate an attacker with knowledge of the distribution they are guessing. While a real attacker wouldn’t have training data from their target distribution, they might be able to tailor their attack by deriving partial knowledge – common locales and other user demographics (RockYou includes many Romanian-language passwords in addition to English), site-specific vocabulary (game terminology, say), and so on.

Our estimators are given ranked lists of common tokens as their training data, with one separately ranked list per data source. NIST and KeePass do not make use of rank, instead performing membership tests on a union of their lists. Rather than attempting to precisely match the training sources supplied to PGS, our estimator sources more closely match those used in the current production version of `zxcvbn`. For example, in the spirit of free software, we avoid the Google Web Corpus which is only available through a license via the Linguistic Data Consortium. Instead of counting top passwords from the MySpace’06 leak, our estimators use the Xato’15 corpus which is over 200 times bigger.

In all, we count top tokens from the PGS training portion of RockYou’09 and Yahoo’12 (test sets are excluded from the count), Xato’15, 1-grams from English Wikipedia, common words from a Wiktionary 29M-word frequency study of US television and film [10], and common names and surnames from the 1990 US Census [1]. Appendix B has more details on our data sources and algorithm implementations.

We experiment with three estimator training set sizes by truncating the frequency lists at three points: 100k (1.52 MB of gzipped storage), 10k (245 kB), and 1k (29.3 kB). In the 10k set, for example, each ranked list longer than 10k tokens is cut off at that point.

Metrics

When PGS is unable to guess a password, we exclude it from our sample set S . On each sampled password $x_i \in S$, we then measure an algorithm’s estimation error by computing its order-of-magnitude difference Δ_i from PGS,

$$\Delta_i = \log_{10} \frac{g_{alg}(x_i)}{g_{pgs}(x_i)} \quad (4)$$

where g_{alg} is the guess attempt estimate of the algorithm and g_{pgs} is the minimum guess order of the four PGS guessing attacks. For example, $\Delta_i = -2$ means the algorithm underestimated guesses by 2 orders of magnitude compared to PGS for password x_i .

We compare PGS to the estimator algorithms in three ways. First, to give a rough sense of the shape of estimator accuracy, we show log-log scatter plots spanning from 10^0 to 10^{15} guesses, with g_{pgs} on the x axis, g_{alg} on the y axis, and a point for every $x_i \in S$. Second, we show the distribution of Δ_i as a histogram by binning values to their nearest multiple of .5, corresponding to half-orders of magnitude. Third, we calculate the following summary statistics:

$$|\Delta| = \frac{1}{|S|} \sum_{i \in S} |\Delta_i| \quad (5)$$

$$\Delta^+ = \frac{1}{|S|} \sum_{i \in S} \begin{cases} \Delta_i & \text{if } \Delta_i \geq 0 \\ 0 & \text{if } \Delta_i < 0 \end{cases} \quad (6)$$

$|\Delta|$ gives a sense of accuracy, equally penalizing under- and overestimation. Δ^+ measures overestimation. Fewer and smaller overestimations improve (reduce) this metric. We calculate summary statistics within an online range $g_{pgs} < 10^6$ and separately higher magnitudes.

One comparison challenge is that KeePass and NIST output entropy as bits, whereas we want to compare algorithms in terms of guesses. While commonplace among password strength estimators, including the 2012 version of `zxcvbn`, it is mathematically improper to apply entropy, a term that applies to distributions, to individual events. Neither KeePass nor NIST formalize the type of entropy they model, so we assume that n bits of strength means guessing the password is equivalent to guessing a value of a random variable X according to

$$n = H(X) = - \sum_i p(x_i) \log_2 p(x_i) \quad (7)$$

Assuming the guesser knows the distribution over X and tries guesses x_i in decreasing order of probability $p(x_i)$, a lower bound on the expected number of guesses $E[G(X)]$ can be shown [41] to be:

$$E[G(X)] \geq 2^{H(X)-2} + 1 \quad (8)$$

provided that $H(X) \geq 2$. We use this conservative lower bound to convert bits into guesses. Had we additionally assumed a uniform distribution, our expected guesses would be $2^{H(X)-1}$, a negligible difference for our logarithmic accuracy comparisons.

5.2 Results

Of the RockYou 15k test set, PGS cracked 39.68% within our online guessing range of up to 10^6 guesses and 52.65% above 10^6 , leaving 7.67% unguessed.

5.2.1 Choice of Algorithm

Figures 1-3 give a sense of how algorithm choice affects guess attempt estimation. The solid diagonal corresponds to $\Delta = 0$, indicating estimator agreement with

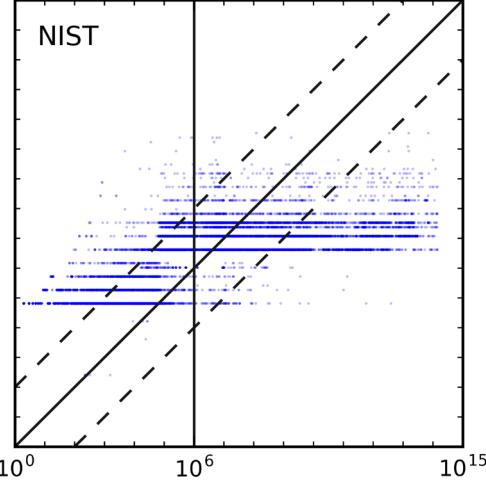


Figure 1: PGS (x axis) vs. NIST (y axis), 100k token set. Points on the solid diagonal indicate agreement between PGS and NIST ($\Delta_i = 0$). Points above the solid diagonal indicate overestimation. Points above the top dashed diagonal indicate overestimation by more than two orders of magnitude ($\Delta_i > 2$).

PGS. Points above the top / below the bottom dotted lines over/underestimate by more than 2 orders of magnitude ($\Delta_i > 2$ above the top line, $\Delta_i < -2$ below the bottom line). Points to the left of $g_{pgs} = 10^6$ indicate samples potentially susceptible to online guessing as argued in Section 3.

NIST and KeePass both exhibit substantial horizontal banding in the low online range. Figure 2 shows that a KeePass estimate of about $10^{4.5}$ can range anywhere from about 10^{-25} to 10^6 PGS guesses. Figure 1 shows that NIST has a similar band at about $10^{4.8}$, and that NIST tends to overestimate in the online range and lean towards underestimation at higher magnitudes. In Table 1 we measured NIST and KeePass to be respectively off by $|\Delta| = 1.81$ and 1.43 orders of magnitude on average within the online range. We conclude neither are suitable for estimating online guessing resistance; however, we expect KeePass could fix its low-order banding problem by incorporating token frequency rank instead of a fixed entropy for each dictionary match.

Figure 3 demonstrates that zxcvbn grows roughly linear with PGS up until about 10^5 , corresponding to the maximum rank of the 100k token set. Both zxcvbn and KeePass suffer from a spike in overestimation approximately between 10^5 and 10^7 . We speculate this is because PGS is trained on 21M unique tokens and, in one attack, tries all of them in order of popularity before moving onto more complex guessing. Hence, the greatest overestimation in both cases happens between the estimator dictionary cutoff and PGS dictionary cutoff, high-

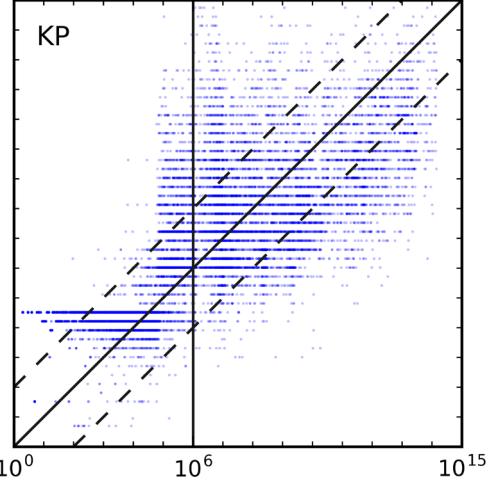


Figure 2: PGS (x) vs. KeePass (y), 100k token set.

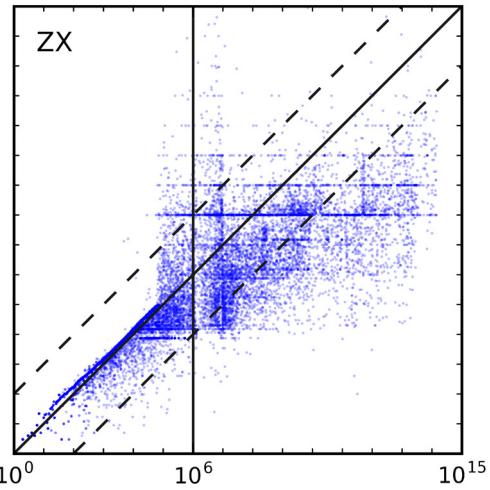


Figure 3: PGS (x) vs. zxcvbn (y), 100k token set.

lighting the sensitivity of estimator dictionary size.

The horizontal banding at fixed orders of magnitude in zxcvbn corresponds to bruteforce matches where no other pattern could be identified. Detailed in Section 4, zxcvbn rewards 10^l guesses for length- l unmatched regions. zxcvbn has comparable but slightly worse $|\Delta|$ and Δ^+ than NIST past the online range. Given both have a low Δ^+ , this primarily demonstrates a usability problem at higher magnitudes (overly harsh feedback).

Figure 4 counts and normalizes Δ_i in bin multiples of .5, demonstrating that zxcvbn is within $\pm .25$ orders of magnitude of PGS about 50% of the time within the online range. The sharp drop-off to the right indicates infrequent overestimation in this range. Figure 4 also shows that, within the online range, NIST and KeePass accuracy could be improved by respectively dividing their estimates by about 10^2 and 10^1 guesses; however, both

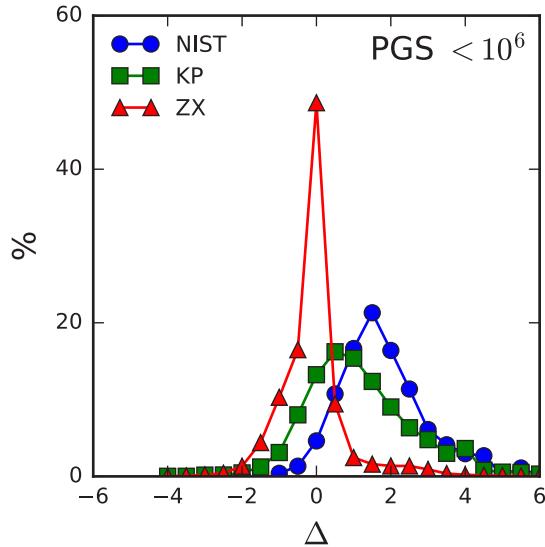


Figure 4: Δ histograms, 100k token set, online attack range ($g_{pgs} < 10^6$). zxvcvbn spikes at $\Delta = 0$ then conservatively falls off to the right.

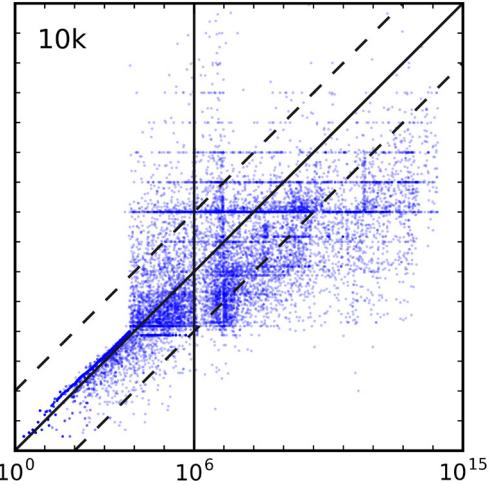


Figure 5: PGS (x) vs. zxvcvbn (y), 10k token set.

would still exhibit substantial overestimation tails.

5.2.2 Choice of Data

We now contrast a single algorithm zxvcvbn with varying data. Figures 3, 5, and 6 show zxvcvbn with the 100k, 10k and 1k token sets. The noticeable effect is linear growth up until 10^5 , 10^4 and 10^3 , respectively. Overestimation is nearly non-existent in these respective ranges.

Within the online range, $|\Delta|$ and Δ^+ noticeably improve with more data. Past the online range, more data makes the algorithm more conservative, with progressively higher $|\Delta|$ and lower Δ^+ .

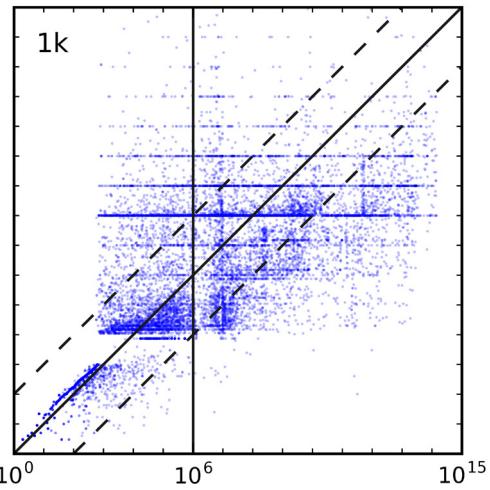


Figure 6: PGS (x) vs. zxvcvbn (y), 1k token set.

5.2.3 Impact of Pattern Matching

We lastly measure the impact of matching additional patterns beyond token lookup. Figure 7 shows a variant of zxvcvbn that recognizes case-insensitive token lookups only. Differences from Figure 3 include noticeably more overestimation before 10^5 and more prominent horizontal banding.

For our $|\Delta|$ and Δ^+ figures, we show the cumulative effect of starting with case-insensitive token matching only, and then incrementally matching additional types of patterns. Overall the impact is small compared to supplying additional data, but the space- and time- cost is near-zero, hence we consider these extra patterns a strict

	PGS < 10^6		PGS > 10^6	
	$ \Delta $	Δ^+	$ \Delta $	Δ^+
NIST-100k	1.81	1.79	2.04	0.14
KP-100k	1.43	1.31	1.81	0.70
ZX-100k	0.58	0.27	2.20	0.21
ZX-1k	1.47	1.23	2.13	0.46
ZX-10k	0.82	0.53	2.18	0.28
ZX-100k	0.58	0.27	2.20	0.21
 ZX-100k:				
tokens only	0.68	0.48	1.85	0.30
+reversed/133t	0.68	0.47	1.87	0.29
+date/year	0.60	0.35	2.13	0.23
+keyboard	0.60	0.34	2.13	0.22
+repeat	0.57	0.28	2.19	0.21
+sequence	0.58	0.27	2.20	0.21

Table 1: $|\Delta|$ and Δ^+ summary statistics. The top, middle and bottom portions correspond to Sections 5.2.1-5.2.3, respectively.

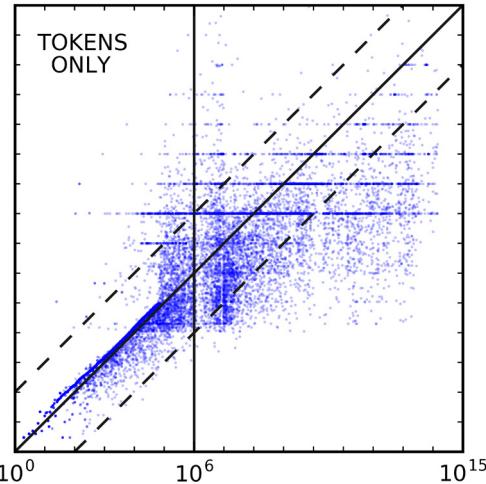


Figure 7: PGS (x) vs. zxcvbn (y), 100k token set, case-insensitive token lookups only. Table 1 shows the cumulative benefit of matching additional patterns.

improvement.

Within the online range, $|\Delta|$ shrinks by 15% after all pattern types are included. Δ^+ shrinks by 44%. As with adding more data, past 10^6 , adding patterns increases estimator conservatism, with a progressively higher $|\Delta|$ and lower Δ^+ .

5.3 Performance Benchmarks

To measure zxcvbn runtime performance, we concatenated the RockYou’09 and Yahoo’12 test sets into a single 30k list. We ran through that sample 1000 times, recording zxcvbn runtime for each password, and finally averaged the runtimes across the batches for each password. We obtained the following runtime percentiles in milliseconds:

	25^{th}	50^{th}	75^{th}	99.9^{th}	max
Chrome (ms)	0.31	0.44	0.60	3.34	27.33
node (ms)	0.38	0.53	0.72	3.00	29.61

We checked that these numbers are comparable to running a single batch, to verify that we avoided caching effects and other interpreter optimizations. Our trials used 64-bit Chrome 48 and 64-bit node v5.5.0 on OS X 10.10.4 running on a late 2013 MacBook Pro with a 2.6 GHz Intel Core i7 Haswell CPU.

5.4 Limitations

Because we measured estimator accuracy against the current best guessing techniques, accuracy will need to be reevaluated as the state of the art advances. By including training and test data from the same distribution, we

erred on the side of aggressiveness for our guessing simulation; however, test data aside, to the extent that PGS and zxcvbn are trained on the same or similar data with the same models, we expect similar accuracy at low magnitudes up until zxcvbn’s frequency rank cutoff (given a harder guessing task, that range might span a lower percentage of the test set). Our experiments measured estimator accuracy but not their influence on password selection behavior; a separate user study would be valuable, with results that would likely depend on how competing estimators are used and presented.

6 Conclusion

To the extent that our estimator is trained on the same or similar password leaks and dictionaries as employed by attackers, we’ve demonstrated that checking the minimum rank over a series of frequency ranked lists, combined with light pattern matching, is enough to accurately and conservatively predict today’s best guessing attacks within the range of an online attack. zxcvbn works entirely client-side, runs in milliseconds, and has a configurable download size: 1.5MB of compressed storage is sufficient for high accuracy up to 10^5 guesses, 245 kB up to 10^4 guesses, or 29 kB up to 10^3 guesses. zxcvbn can be bundled with clients or asynchronously downloaded on demand. It works as-is on most browsers, browser extensions, Android, iOS, and server-side javascript frameworks, with ports available in several other major languages. In place of LUDS, it is our hope that client-side estimators such as zxcvbn will increasingly be deployed to allow more flexibility for users and better security guarantees for adopters.

Acknowledgments

I’d like to thank Tom Ristenpart for shepherding this paper as well as the anonymous reviewers for their helpful comments. Thanks to Blase Ur, Sean Segreti, Henry Dixon, and the rest of the Password Research Group for their advice and help running the Password Guessability Service. Thanks to Mark Burnett for his password corpus. Thanks to Rian Hunter for extracting the KeePass estimator into a standalone Mono binary. Thanks to Devdatta Akhawe, Andrew Bortz, Hongyi Hu, Anton Mityagin, Brian Smith, and Josh Lifton for their feedback and guidance. Last but not least, a big thanks to Dropbox for sponsoring this research.

Availability

zxcvbn is free software under the MIT License:

<http://github.com/dropbox/zxcvbn>

References

- [1] Frequently Occurring Surnames from Census 1990. http://www.census.gov/topics/population/genealogy/data/1990_census/1990_census_namefiles.html.
- [2] Hashcat advanced password recovery. <http://hashcat.net>.
- [3] John the Ripper password cracker. <http://www.openwall.com/john>.
- [4] KeePass Help Center: Password Quality Estimation. http://keepass.info/help/kb/pw_quality_est.html.
- [5] KeePass Password Safe. <http://keepass.info>.
- [6] Penn Treebank tokenization. <http://www.cis.upenn.edu/~treebank/tokenization.html>.
- [7] RockYou Hack: From Bad To Worse. <http://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords>.
- [8] The CMU Password Research Group's Password Guessability Service. <https://pgs.ece.cmu.edu/>.
- [9] The InCommon Assurance Program. <https://incommon.org/assurance>.
- [10] Wiktionary: Frequency lists. https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists.
- [11] Yahoo hacked, 450,000 passwords posted online. <http://www.cnn.com/2012/07/12/tech/web/yahoo-users-hacked>.
- [12] Password management guideline. CSC-STD-002-85. U.S. Department of Defense, May 1985.
- [13] Password usage. *Federal Information Processing Standards Publication 112*. U.S. National Institute of Standards and Technology, April 1985.
- [14] Q3 2015 State Of The Internet. Akamai Technologies, December 2015.
- [15] BERGADANO, F., CRISPO, B., AND RUFFO, G. Proactive password checking with decision trees. In *4th ACM Conference on Computer and Communications Security* (New York, NY, USA, 1997), CCS '97, ACM, pp. 67–77.
- [16] Biryukov, A., Dinu, D., and Khovalovich, D. Argon2: New generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy* (March 2016), pp. 292–302.
- [17] BISHOP, M. Anatomy of a proactive password changer. In *3rd UNIX Security Symposium* (Berkeley, CA, USA, Sep. 1992), USENIX, pp. 171–184.
- [18] BLUNDO, C., D'ARCO, P., DE SANTIS, A., AND GALDI, C. Hypocrates: a new proactive password checker. *Journal of Systems and Software* 71, 1 (2004), 163–175.
- [19] BONNEAU, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 538–552.
- [20] BONNEAU, J., HERLEY, C., OORSCHOT, P. C. v., AND STAJANO, F. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), IEEE Computer Society, pp. 553–567.
- [21] BONNEAU, J., AND PREIBUSCH, S. The password thicket: Technical and market failures in human authentication on the web. In *WEIS* (2010).
- [22] BURR, W., DODSON, D., NEWTON, E., PERLNER, R., POLK, T., GUPTA, S., AND NABBUS, E. NIST Special Publication 800-63-2 Electronic Authentication Guideline. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, August 2013.
- [23] CALIFA, J. Patronizing passwords. <http://joelcalifa.com/blog/patronizing-passwords>, 2015.
- [24] CASTELLUCCIA, C., DÜRMUTH, M., AND PERITO, D. Adaptive password-strength meters from markov models. In *NDSS* (2012), The Internet Society.
- [25] DE CARNÉ DE CARNAVALET, X., AND MANNAN, M. A large-scale evaluation of high-impact password strength meters. *ACM Transactions on Information and System Security (TISSEC)* 18, 1 (2015).
- [26] DELL'AMICO, M., AND FILIPPONE, M. Monte Carlo strength evaluation: Fast and reliable password checking. In *22nd ACM Conference on Computer and Communications Security* (Denver, CO, USA, October 2015).
- [27] DELL'AMICO, M., MICHARDI, P., AND ROUDIER, Y. Password strength: An empirical analysis. In *INFOCOM, 2010 Proceedings IEEE* (March 2010), pp. 1–9.
- [28] EGELMAN, S., SOTIRAKOPOULOS, A., MUSLUKHOV, I., BEZNOSOV, K., AND HERLEY, C. Does my password go up to eleven? the impact of password meters on password selection. In *SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2013), CHI '13, ACM, pp. 2379–2388.
- [29] FLORÊNCIO, D., AND HERLEY, C. A large-scale study of web password habits. In *16th international conference on the World Wide Web* (May 2007), ACM, pp. 657–666.
- [30] FLORÊNCIO, D., HERLEY, C., AND OORSCHOT, P. C. v. An administrator's guide to internet password research. In *28th USENIX Conference on Large Installation System Administration* (Berkeley, CA, USA, 2014), LISA'14, USENIX, pp. 35–52.
- [31] HERLEY, C. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *New Security Paradigms Workshop* (2009), ACM, pp. 133–144.
- [32] HERLEY, C., AND OORSCHOT, P. C. v. A research agenda acknowledging the persistence of passwords. *2012 IEEE Symposium on Security and Privacy* 10, 1 (Jan 2012), 28–36.
- [33] HOLLY, R. Project Abacus is an ATAP project aimed at killing the password. *Android Central*, May 2015.
- [34] INGLENT, P. G., AND SASSE, M. A. The true cost of unusable password policies: Password use in the wild. In *SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2010), CHI '10, ACM, pp. 383–392.
- [35] KLEIN, D. V. Foiling the cracker: A survey of, and improvements to, password security. In *2nd USENIX Security Workshop* (1990), pp. 5–14.
- [36] KOMANDURI, S., BAUER, L., CHRISTIN, N., AND OORSCHOT, P. C. v. *Modeling the adversary to evaluate password strength with limited samples*. PhD thesis, Carnegie Mellon University, 2015.
- [37] KOMANDURI, S., SHAY, R., CRANOR, L. F., HERLEY, C., AND SCHECHTER, S. Telepathwords: Preventing weak passwords by reading users' minds. In *23rd USENIX Security Symposium* (2014), pp. 591–606.
- [38] LI, Z., HAN, W., AND XU, W. A large-scale empirical analysis of chinese web passwords. In *23rd USENIX Security Symposium* (San Diego, CA, Aug. 2014), USENIX, pp. 559–574.
- [39] MA, J., YANG, W., LUO, M., AND LI, N. A study of probabilistic password models. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 689–704.
- [40] MANBER, U., AND WU, S. An algorithm for approximate membership checking with application to password security. *Information Processing Letters* 50, 4 (1994), 191–197.
- [41] MASSEY, J. L. Guessing and entropy. In *IEEE International Symposium on Information Theory* (1994), IEEE, p. 204.

- [42] MELICHER, W., UR, B., SEGRETI, S. M., KOMANDURI, S., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Fast, lean, and accurate: Modeling password guessability using neural networks. In *Proceedings of the 25th USENIX Security Symposium* (Aug. 2016). To appear.
- [43] MORRIS, R., AND THOMPSON, K. Password security: A case history. *Communications of the ACM* 22, 11 (1979), 594–597.
- [44] MUNROE, R. xkcd: password strength. <https://xkcd.com/936/>, August 2011.
- [45] NAGLE, J. An Obvious Password Detector. <http://securitydigest.org/phage/archive/240>, November 1988.
- [46] SCHECHTER, S., HERLEY, C., AND MITZENMACHER, M. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *5th USENIX Conference on Hot Topics in Security* (Berkeley, CA, USA, 2010), HotSec’10, USENIX, pp. 1–8.
- [47] SHANNON, C. E. A mathematical theory of communication. In *The Bell System Technical Journal* (1948), vol. 27, pp. 379–423, 623–656.
- [48] SHAY, R., KOMANDURI, S., KELLEY, P. G., LEON, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Encountering stronger password requirements: User attitudes and behaviors. In *6th Symposium on Usable Privacy and Security* (New York, NY, USA, 2010), SOUPS ’10, ACM, pp. 2:1–2:20.
- [49] SPAFFORD, E. H. OPUS: Preventing weak password choices. *Computers & Security* 11, 3 (May 1992), 273–278.
- [50] UR, B., KELLEY, P. G., KOMANDURI, S., LEE, J., MAASS, M., MAZUREK, M. L., PASSARO, T., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. How does your password measure up? the effect of strength meters on password creation. In *21st USENIX Security Symposium* (Bellevue, WA, 2012), USENIX, pp. 65–80.
- [51] UR, B., SEGRETI, S. M., BAUER, L., CHRISTIN, N., CRANOR, L. F., KOMANDURI, S., KURIOVA, D., MAZUREK, M. L., MELICHER, W., AND SHAY, R. Measuring real-world accuracies and biases in modeling password guessability. In *24th USENIX Security Symposium* (2015), pp. 463–481.
- [52] WANG, D., AND WANG, P. The emperor’s new password creation policies. In *Computer Security, ESORICS 2015*. Springer, 2015, pp. 456–477.
- [53] WEIR, M., AGGARWAL, S., COLLINS, M., AND STERN, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS ’10, ACM, pp. 162–175.
- [54] WEIR, M., AGGARWAL, S., DE MEDEIROS, B., AND GLODEK, B. Password cracking using probabilistic context-free grammars. In *2009 IEEE Symposium on Security and Privacy* (May 2009), pp. 391–405.
- [55] WHEELER, D. zxcvbn: Realistic password strength estimation. Dropbox Tech Blog, 2012.
- [56] YAN, J. J. A note on proactive password checking. In *2001 Workshop on New Security Paradigms* (2001), ACM, pp. 127–135.

A zxcvbn vs. 3class8

We made the claim that zxcvbn is no harder to adopt than LUDS strategies such as 3class8. We provided a CommonJS implementation in

Section 3 that rejects passwords guessable in 500 attempts according to zxcvbn. For comparison, here we provide our implementation of 3class8 back-to-back with equivalent zxcvbn integrations using two other common JavaScript module interfaces: global namespacing and Asynchronous Module Definition with RequireJS.

```
var meets_3class8 = function(password) {
  var classes = 0;
  if /[a-z]/.test(password) {classes++;}
  if /[A-Z]/.test(password) {classes++;}
  if /\d/.test(password) {classes++;}
  if /[\\W_]/.test(password) {classes++;}
  return classes >= 3 and password.length > 8;
}

// in .html: <script src="zxcvbn.js"></script>
var meets_policy_global = function(password) {
  return zxcvbn(password).guesses > 500;
};

requirejs(["path/to/zxcvbn"], function(zxcvbn) {
  var meets_policy_amd = function(password) {
    return zxcvbn(password).guesses > 500;
  };
});
```

B Experiment implementation details

For the sake of reproducibility, we detail the specifics of the algorithms and data we employed in our experiments.

Algorithms

KeePass: We downloaded the C# source of KeePass 2.31 released on 1/9/2016 and extracted its strength estimator into a stand-alone Mono executable that takes a token dictionary as input.

NIST: calculated as specified in Section 2. The NIST 2013 guideline [22] does not precisely define the dictionary check but recommends applying common word transformations. We ignore case and check for reversed words and common 133t substitutions, the same as in zxcvbn. NIST specifies awarding up to 6 bits for passing the dictionary check, decreasing to 0 bits at or above 20 characters, but doesn’t otherwise specify how to award bits. We award a full 6 bits for passwords at or under 10 characters, 4 bits if between 11 and 15, and otherwise 2 bits. NIST recommends a dictionary of at least 50k tokens. The 100k token set described in Section 5 consists of about 390k unique tokens (consisting of several lists ending up to rank-100k).

zxcvbn: Outlined in detail in Section 4.

Data

Within each data source, all tokens were lowercased, counted, and sorted by descending count. When multiple lists contained the same token, that token was filtered from every list but the one with the lowest (most popular) rank. We made use of the following raw data:

RockYou: 32M passwords leaked in 2009 [7], excluding a random 15k test set consisting of passwords of 8 or more characters.

Yahoo: 450k passwords leaked in 2012 [11], excluding a random 15k test set consisting of passwords of 8 or more characters. We cut this list off at 10k top tokens, given the smaller size of the leak.

Xato: Mark Burnett’s 10M password corpus, released in 2015 on Xato.net and compiled by sampling thousands of password leaks

over approximately 15 years. These passwords mostly appear to be from Western users. The authors confirmed with Burnett that the RockYou set is not sampled in Xato; however, Xato likely includes a small number of samples from the Yahoo set. Given the relative sizes of the two sets, Yahoo could at most make up 4.5% of Xato; however, we expect a much smaller percentage from talking to Mark.

Wikipedia: 1-grams from the English Wikipedia database dump of 2015-10-2. We include all Wikipedia articles but not previous revisions, edit histories, template files, or metadata. We parse text from wiki markup via the open-source `WikiExtractor.py` and tokenize according to the Penn Treebank method [6].

Wiktionary: Words from a 2006 Wiktionary word frequency study counting 29M words from US television and film [10]. This list balances Wikipedia’s formal English with casual language and slang. 40k unique tokens.

USCensus: Names and surnames from the 1990 United States Census [1] ranked by frequency as three separate lists: surnames, female names, and male names. We cut surnames off at 10k tokens.

C Yahoo Analysis

For reference, we reproduce the results of Section 5.2 with a sample of Yahoo’12 passwords instead of RockYou’09. Of the 15k test set, PGS cracked 29.05% within our Section 3 online guessing cutoff at 10^6 guesses and 60.07% above 10^6 , leaving 10.88% unguessed.

Choice of Algorithm

Figures 8-10 respectively show PGS vs NIST, KeePass, and zxvcvbn, with each estimator supplied with the same 100k token set. As in the RockYou sample, NIST and KeePass exhibit substantial horizontal banding and overestimate at low magnitudes. At higher magnitudes, NIST tends to underestimate.

Figure 10 demonstrates that zxvcvbn grows roughly linear with PGS, leaning towards underestimation, up until 10^5 guesses. Observable in the RockYou sample but more pronounced here, KeePass and zxvcvbn both experience a spike in overestimation between 10^5 and 10^7 . We offer the same explanation as with RockYou: PGS is trained on a little over 10^7 unique tokens, some of which are long and unrecognized by the estimators. PGS occasionally succeeds making single-token guesses at these higher magnitudes, leading to a spike in inaccuracy between estimator dictionary cutoff and PGS dictionary cutoff.

In Figure 11, we see a similar Δ_i spike at zero for zxvcvbn followed by a sharp decline to the right, indicating high accuracy and low overestimation within an online range.

Choice of Data

Figures 12-13 show PGS vs. zxvcvbn with 10k and 1k token sets, respectively. We observe the same noticeable effect as with RockYou: high accuracy at low magnitudes up until the max token rank cutoff at 10^4 and 10^3 , respectively. Referring to Table 2, $|\Delta|$ and Δ^+ noticeably improve with more data within the online range. Past the online range, more data makes the algorithm more conservative, with progressively higher $|\Delta|$ and lower Δ^+ .

Impact of Pattern Matching

Figure 14 shows a variant of zxvcvbn, supplied with the 100k token set, that matches case-insensitive token lookups only. We similarly observe more overestimation before $g_{pgs} = 10^5$ and more prominent horizontal banding at higher magnitudes compared to Figure 10.

The bottom portion of Table 2 shows the cumulative effect of matching additional pattern types. Within the online range, $|\Delta|$ and Δ^+ shrink by about 5% and 46%, respectively.

	PGS < 10^6		PGS > 10^6	
	$ \Delta $	Δ^+	$ \Delta $	Δ^+
NIST-100k	1.46	1.43	2.19	0.13
KP-100k	1.24	1.05	1.85	0.83
ZX-100k	0.74	0.19	2.45	0.26
ZX-1k	0.74	0.19	2.45	0.26
ZX-10k	0.91	0.39	2.40	0.32
ZX-100k	1.42	1.04	2.28	0.50
ZX-100k:				
tokens only	0.78	0.35	2.13	0.33
+reversed/133t	0.79	0.33	2.19	0.32
+date/year	0.74	0.26	2.35	0.28
+keyboard	0.74	0.25	2.36	0.27
+repeat	0.73	0.19	2.43	0.26
+sequence	0.74	0.19	2.45	0.26

Table 2: $|\Delta|$ and Δ^+ summary statistics.

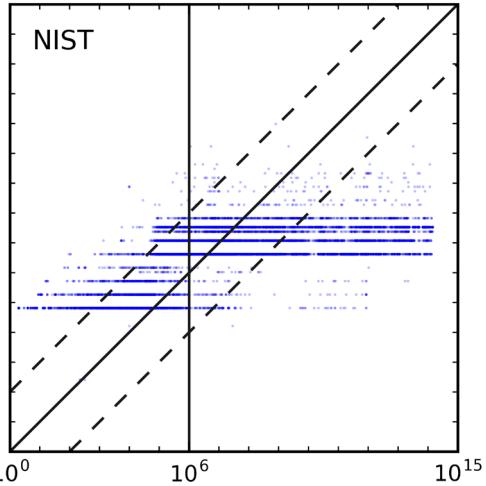


Figure 8: PGS (x) vs. NIST (y), 100k token set.

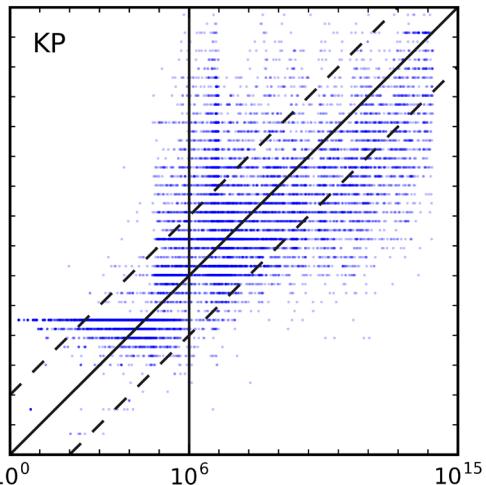


Figure 9: PGS (x) vs. KeePass (y), 100k token set.

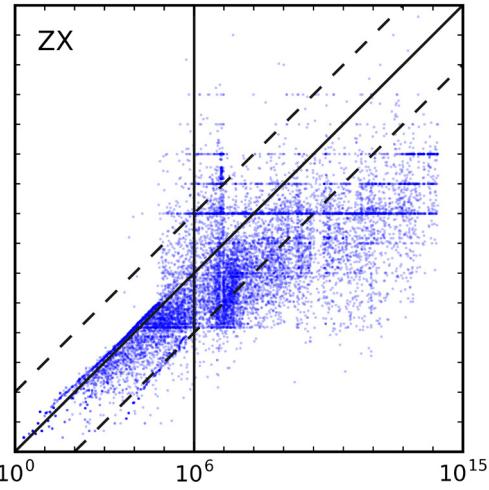


Figure 10: PGS (x) vs. zxvcvbn (y), 100k token set.

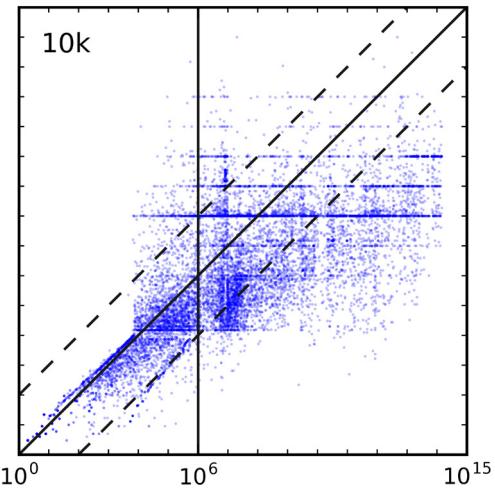


Figure 12: PGS (x) vs. zxvcvbn (y), 10k token set.

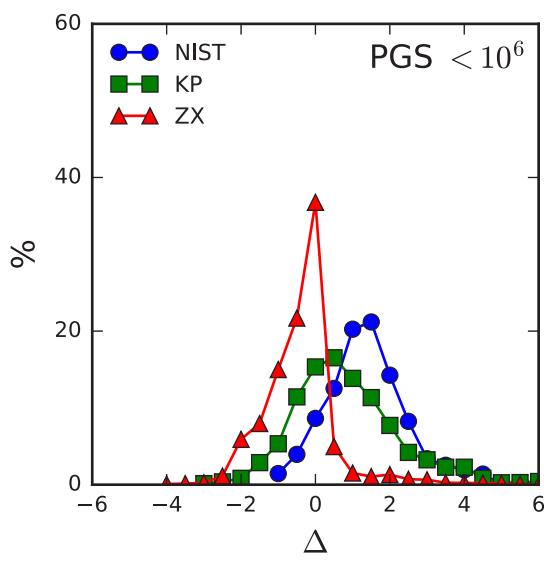


Figure 11: Δ histograms, 100k token set, online attack range ($g_{pgs} < 10^6$). zxvcvbn spikes at $\Delta = 0$ then conservatively falls off to the right.

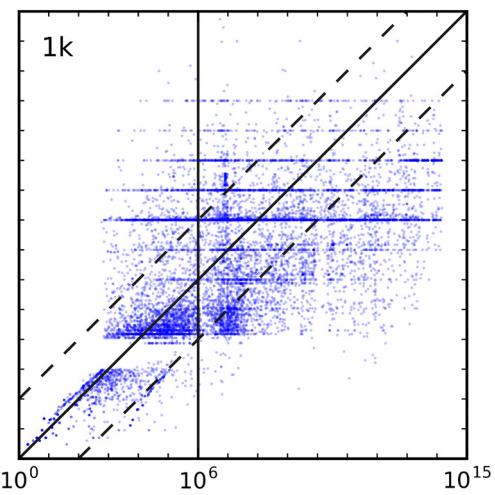


Figure 13: PGS (x) vs. zxvcvbn (y), 1k token set.

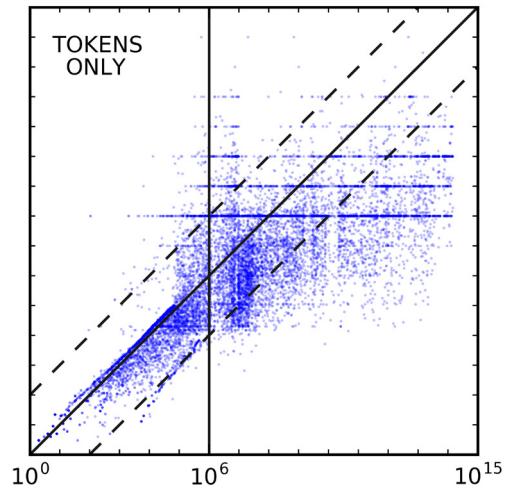


Figure 14: PGS (x) vs. zxvcvbn (y), 100k token set, case-insensitive token lookups only.

Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks

William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri,
Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor
Carnegie Mellon University

Abstract

Human-chosen text passwords, today’s dominant form of authentication, are vulnerable to guessing attacks. Unfortunately, existing approaches for evaluating password strength by modeling adversarial password guessing are either inaccurate or orders of magnitude too large and too slow for real-time, client-side password checking. We propose using artificial neural networks to model text passwords’ resistance to guessing attacks and explore how different architectures and training methods impact neural networks’ guessing effectiveness. We show that neural networks can often guess passwords more effectively than state-of-the-art approaches, such as probabilistic context-free grammars and Markov models. We also show that our neural networks can be highly compressed—to as little as hundreds of kilobytes—without substantially worsening guessing effectiveness. Building on these results, we implement in JavaScript the first principled client-side model of password guessing, which analyzes a password’s resistance to a guessing attack of arbitrary duration with sub-second latency. Together, our contributions enable more accurate and practical password checking than was previously possible.

1 Introduction

Text passwords are currently the most common form of authentication, and they promise to continue to be so for the foreseeable future [53]. Unfortunately, users often choose predictable passwords, enabling password-guessing attacks. In response, proactive password checking is used to evaluate password strength [19].

A common way to evaluate the strength of a password is by running or simulating password-guessing techniques [35, 59, 92]. A suite of well-configured guessing techniques, encompassing both probabilistic approaches [37, 65, 93] and off-the-shelf password-recovery tools [74, 83], can accurately model the vulnerability of

passwords to guessing by expert attackers [89]. Unfortunately, these techniques are often very computationally intensive, requiring hundreds of megabytes to gigabytes of disk space, and taking days to execute. Therefore, they are typically unsuitable for real-time evaluation of password strength, and sometimes for any practically useful evaluation of password strength.

With the goal of gauging the strength of human-chosen text passwords both more accurately and more practically, we propose using artificial neural networks to guess passwords. Artificial neural networks (hereafter referred to as “neural networks”) are a machine-learning technique designed to approximate highly dimensional functions. They have been shown to be very effective at generating novel sequences [49, 84], suggesting a natural fit for generating password guesses.

In this paper, we first comprehensively test the impact of varying the neural network model size, model architecture, training data, and training technique on the network’s ability to guess different types of passwords. We compare our implementation of neural networks to state-of-the-art password-guessing models, including widely studied Markov models [65] and probabilistic context-free grammars [59, 93], as well as software tools using mangled dictionary entries [74, 83]. In our tests, we evaluate the performance of probabilistic models to large numbers of guesses using recently proposed Monte Carlo methods [34]. We find that neural networks guess passwords more successfully than other password-guessing methods in general, especially so beyond 10^{10} guesses and on non-traditional password policies. These cases are interesting because password-guessing attacks often proceed far beyond 10^{10} guesses [44, 46] and because existing password-guessing attacks underperform on new, non-traditional password policies [79, 80].

Although more effective password guessing using neural networks is an important contribution on its own, we also show that the neural networks we use can be highly compressed with minimal loss of guessing ef-

fectiveness. Our approach is thus far more suitable than existing password-guessing methods for client-side password checking. Most existing client-side password checkers are inaccurate [33] because they rely on simple, easily compressible heuristics, such as counting the number of characters or character classes in a password. In contrast, we show that a highly compressed neural network more accurately measures password strength than existing client-side checkers. We can compress such a neural network into hundreds of kilobytes, which is small enough to be included in an app for mobile devices, bundled with encryption software, or used in a web page password meter.

To demonstrate the practical suitability of neural networks for client-side password checking, we implement and benchmark a neural-network password checker in JavaScript. This implementation, which we have released as open-source software,¹ is immediately suitable for use in mobile apps, browser extensions, and web page password meters. Our implementation gives real-time feedback on password strength in fractions of a second, and it more accurately measures resistance to guessing than existing client-side methods.

In summary, this paper makes three main contributions that together substantially increase our ability to detect and help eliminate weak passwords. First, we propose neural networks as a model for guessing human-chosen passwords and comprehensively evaluate how varying their training, parameters, and compression impacts guessing effectiveness. In many circumstances, neural networks guess more accurately than state-of-art techniques. Second, leveraging neural networks, we create a password-guessing model sufficiently compressible and efficient for client-side proactive password checking. Third, we build and benchmark a JavaScript implementation of such a checker. In common web browsers running on commodity hardware, this implementation models an arbitrarily high number of adversarial guesses with sub-second latency, while requiring only hundreds of kilobytes of data to be transferred to a client. Together, our contributions enable more accurate proactive password checking, in a far broader range of common scenarios, than was previously possible.

2 Background and Related Work

To highlight when password strength matters, we first summarize password-guessing attacks. We then discuss metrics and models for evaluating password strength, as well as lightweight methods for estimating password strength during password creation. Finally, we summarize prior work on generating text using neural networks.

¹https://github.com/cupslab/neural_network_cracking

2.1 Password-Guessing Attacks

The extent to which passwords are vulnerable to guessing attacks is highly situational. For phishing attacks, keyloggers, or shoulder surfing, password strength does not matter. Some systems implement rate-limiting policies, locking an online account or a device after a small number of incorrect attempts. In these cases, passwords other than perhaps the million most predictable are unlikely to be guessed [39].

Guessing attacks are a threat, however, in three other scenarios. First, if rate limiting is not properly implemented, as is believed to have been the case in the 2014 theft of celebrities’ personal photos from Apple’s iCloud [50], large-scale guessing becomes possible. Second, if a database of hashed passwords is stolen, which sadly occurs frequently [20, 23, 27, 45, 46, 67, 73, 75, 87], an offline attack is possible. An attacker chooses likely candidate passwords, hashes them, and searches the database for a matching hash. When a match is found, attackers can rely on the high likelihood of password reuse across accounts and try the same credentials on other systems [32]. Attacks leveraging password reuse have real-world consequences, including the recent compromise of Mozilla’s Bugzilla database due to an administrator reusing a password [76] and the compromise of 20 million accounts on Taobao, a Chinese online shopping website similar to eBay, due to password reuse [36].

Third, common scenarios in which cryptographic key material is derived from, or protected by, a password are vulnerable to large-scale guessing in the same way as hashed password databases for online accounts. For instance, for password managers that sync across devices [52] or privacy-preserving cloud backup tools (e.g., SpiderOak [82]), the security of files stored in the cloud depends directly on password strength. Furthermore, cryptographic keys used for asymmetric secure messaging (e.g., GPG private keys), disk-encryption tools (e.g., TrueCrypt), and Windows Domain Kerberos Tickets [31] are protected by human-generated passwords. If the file containing this key material is compromised, the strength of the password is critical for security. The importance of this final scenario is likely to grow with the adoption of password managers and encryption tools.

2.2 Measuring Password Strength

Models of password strength often take one of two conceptual forms. The first relies on purely statistical methods, such as Shannon entropy or other advanced statistical approaches [21, 22]. However, because of the unrealistically large sample sizes required, we consider these types of model out of scope. The second conceptual approach is to simulate adversarial password guess-

ing [34, 65, 89]. Our application of neural networks follows this method. Below, we describe the password-guessing approaches that have been widely studied in academia and used in adversarial password cracking, all of which we compare to neural networks in our analyses. Academic studies of password guessing have focused on probabilistic methods that take as input large password sets, then output guesses in descending probability order. Password cracking tools rely on efficient heuristics to model common password characteristics.

Probabilistic Context-Free Grammars One probabilistic method uses probabilistic context-free grammars (PCFGs) [93]. The intuition behind PCFGs is that passwords are built with template structures (e.g., 6 letters followed by 2 digits) and terminals that fit into those structures. A password’s probability is the probability of its structure multiplied by those of its terminals.

Researchers have found that using separate training sources for structures and terminals improves guessing [59]. It is also beneficial to assign probabilities to unseen terminals by smoothing, as well as to augment guesses generated by the grammar with passwords taken verbatim from the training data without abstracting them into the grammar [60]. Furthermore, using natural-language dictionaries to instantiate terminals improves guessing, particularly for long passwords [91].

Markov Models Using Markov models to guess passwords, first proposed in 2005 [70], has recently been studied more comprehensively [37, 65]. Conceptually, Markov models predict the probability of the next character in a password based on the previous characters, or context characters. Using more context characters can allow for better guesses, yet risks overfitting. Smoothing and backoff methods compensate for overfitting. Researchers have found that a 6-gram Markov model with additive smoothing is often optimal for modeling English-language passwords [65]. We use that configuration in our analyses.

Mangled Wordlist Methods In adversarial password cracking, software tools are commonly used to generate password guesses [44]. The most popular tools transform a wordlist (passwords and dictionary entries) using mangling rules, or transformations intended to model common behaviors in how humans craft passwords. For example, a mangling rule may append a digit and change each ‘a’ to ‘@’. Two popular tools of this type are Hashcat [83] and John the Ripper (JtR, [74]). While these approaches are not directly based on statistical modeling, they produce fairly accurate guesses [89] quickly, which has led to their wide use [44].

2.3 Proactive Password Checking

Although the previously discussed password-guessing models can accurately model human-created passwords [89], they take hours or days and megabytes or gigabytes of disk space, making them too resource-intensive to provide real-time feedback to users. Current real-time password checkers can be categorized based on whether they run entirely client-side. Checkers with a server-side component can be more accurate because they can leverage large amounts of data. For instance, researchers have proposed using server-side Markov models to gauge password strength [26]. Others have studied using training data from leaked passwords and natural-language corpora to show users predictions about what they will type next [61].

Unfortunately, a server-side component introduces substantial disadvantages for security. In some cases, sending a password to a server for password checking destroys all security guarantees. For instance, passwords that protect an encrypted volume (e.g., TrueCrypt) or cryptographic keys (e.g., GPG), as well as the master password for a password manager, should never leave the user’s device, even for proactive password checking. As a result, accurate password checking is often missing from these security-critical applications. In cases when a password is eventually sent to the server (e.g., for an online account), a real-time, server-side component both adds latency and opens password meters to powerful side-channel attacks based on keyboard timing, message size, and caching [81].

Prior client-side password checkers, such as those running entirely in a web browser, rely on heuristics that can be easily encoded. Many common meters rate passwords based on their length or inclusion of different character classes [33, 88]. Unfortunately, in comprehensive tests of both client- and server-side password meters, all but one meter was highly inaccurate [33]. Only zxcvbn [94, 95], which uses dozens of more advanced heuristics, gave reasonably accurate strength estimations. Such meters, however, do not directly model adversarial guessing because of the inability to succinctly encode models and calculate real-time results. In contrast, our approach models adversarial guessing entirely on the client side.

2.4 Neural Networks

Neural networks, which we use to model passwords, are a machine-learning technique for approximating highly dimensional functions. Designed to model human neurons, they are particularly adept at fuzzy classification problems and generating novel sequences. Our method of generating candidate password guesses draws heavily on previous work that generated the probability of

the next element in a string based on the preceding elements [49, 84]. For example, in generating the string *password*, a neural network might be given *passwor* and output that *d* has a high probability of occurring next.

Although password creation and text generation are conceptually similar, little research has attempted to use insights from text generation to model passwords. A decade ago, neural networks were proposed as a method for classifying passwords into two very broad categories (weak or strong) [30], but that work did not seek to model the order in which passwords would be guessed or other aspects of a guessing attack. To our knowledge, the only proposal to use neural networks in a password-guessing attack was a recent blog post [71]. In sharp contrast to our extensive testing of different parameters to make neural networks effective in practice, that work made few refinements to the application of neural networks, leading the author to doubt that the approach has “any practical relevance.” Additionally, that work sought only to model a few likely password guesses, as opposed to our use of Monte Carlo methods to simulate an arbitrary number of guesses.

Conceptually, neural networks have advantages over other methods. In contrast to PCFGs and Markov models, the sequences generated by neural networks can be inexact, novel sequences [49], which led to our intuition that neural networks might be appropriate for password guessing. Prior approaches to probabilistic password guessing (e.g., Markov models [26]) were sufficiently memory-intensive to be impractical on only the client-side. However, neural networks can model natural language in far less space than Markov models [68]. Neural networks have also been shown to transfer knowledge about one task to related tasks [97]. This is crucial for targeting novel password-composition policies, for which training data is sparse at best.

3 System Design

We experimented with a broad range of options in a large design space and eventually arrived at a system design that 1) leverages neural networks for password guessing, and 2) provides a client-side guess estimation method.

3.1 Measuring Password Strength

Similarly to Markov models, neural networks in our system are trained to generate the next character of a password given the preceding characters of a password. Figure 1 illustrates our construction. Like in Markov models [34, 65], we rely on a special password-ending symbol to model the probability of ending a password after a sequence of characters. For example, to calculate the probability of the entire password ‘bad’, we would

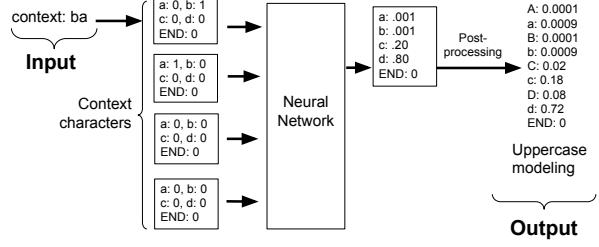


Figure 1: **An example of using a neural network to predict the next character of a password fragment.** The network is being used to predict a ‘d’ given the context ‘ba’. This network uses four characters of context. The probabilities of each next character are the output of the network. Post processing on the network can infer probabilities of uppercase characters.

start with an empty password, and query the network for the probability of seeing a ‘b’, then seeing an ‘a’ after ‘b’, and then of seeing a ‘d’ after ‘ba’, then of seeing a complete password after ‘bad’. To generate passwords from a neural network model, we enumerate all possible passwords whose probability is above a given threshold using a modified beam-search [64], a hybrid of depth-first and breadth-first search. If necessary, we can suppress the generation of non-desirable passwords (e.g., those against the target password policy) by filtering those passwords. Then, we sort passwords by their probability. We use beam-search because breadth-first’s memory requirements do not scale, and because it allows us to take better advantage of GPU parallel processing power than depth-first search. Fundamentally, this method of guess enumeration is similar to that used in Markov models, and it could benefit from the same optimizations, such as approximate sorting [37]. A major advantage over Markov models is that the neural network model can be efficiently implemented on the GPU.

Calculating Guess Numbers In evaluating password strength by modeling a guessing attack, we calculate a password’s *guess number*, or how many guesses it would take an attacker to arrive at that password if guessing passwords in descending order of likelihood. The traditional method of calculating guess numbers by enumeration is computationally intensive. For example, enumerating more than 10^{10} passwords would take roughly 16 days in our unoptimized implementation on an NVidia GeForce GTX 980 Ti. However, in addition to guess number enumeration, we can also estimate guess numbers accurately and efficiently using Monte Carlo simulations, as proposed by Dell’Amico and Filippone [34].

3.2 Our Approach

There are many design decisions necessary to train neural networks. The design space forces us to decide on

the modeling alphabet, context size, type of neural network architecture, training data, and training methodology. We experiment along these dimensions.

Model Architectures In this work, we use recurrent neural networks because they have been shown to be useful for generating text in the context of character-level natural language [49, 84]. Recurrent neural networks are a specific type of neural network where connections in the network can process elements in sequences and use an internal memory to remember information about previous elements in the sequence. We experiment with two different recurrent architectures in Section 5.1.

Alphabet Size We focus on character-level models, rather than more common word-level models, because there is no established dictionary of words for password generation. We also complement our analysis with exploratory experiments using syllable-level models in Section 5.1. We decided to explore hybrid models based on prior work in machine learning [68]. In the hybrid construction, in addition to characters, the neural network is allowed to model sub-word units, such as syllables or tokens. We chose to model 2,000 different tokens based on prior work [68] and represent those tokens the same way we would characters. A more thorough study of tokenized models would explore both more and fewer tokens. Using tokenized structures, the model can then output the probability of the next character being an ‘a’ or the token ‘pass’. We generated the list of tokens by tokenizing words in our training set along character-class boundaries and selecting the 2,000 most frequent ones.

Like prior work [26], we observed empirically that modeling all characters unnecessarily burdens the model and that some characters, like uppercase letters and rare symbols, are better modeled outside of the neural network. We can still create passwords with these characters by interpreting the model’s output as templates. For example, when the neural network predicts an ‘A’ character, we post-process the prediction to predict both ‘a’ and ‘A’ by allocating their respective probabilities based on the number of occurrences of ‘a’ and ‘A’ in the training data—as shown in Figure 1. The intuition here is that we can reduce the amount of resources consumed by the neural network when alternate heuristic approaches can efficiently model certain phenomena (e.g., shifts between lowercase and uppercase letters).

Password Context Predictions rely on the context characters. For example, in Figure 1, the context characters are ‘ba’ and the target prediction is ‘d’. Increasing the number of context characters increases the training

time, while decreasing the number of context characters could potentially decrease guessing success.

We experimented with using all previous characters in the password as context and with only using the previous ten characters. We found in preliminary tests that using ten characters was as successful at guessing and trained up to an order of magnitude faster, and thus settled on this choice. When there are fewer than ten context characters, we pad the input with zeros. In comparison, best-performing Markov models typically use five characters of context [34, 65]. While Markov models can overfit if given too much context, neural networks typically overfit when there are too many parameters.

Providing context characters in reverse order—e.g., predicting ‘d’ from ‘rowssap’ instead of ‘passwor’—has been shown to sometimes improve performance [48]. We empirically evaluate this technique in Section 5.1.

Model Size We must also decide how many parameters to include in models. To gauge the effect of changing the model size on guessing success, we test a large neural network with 15,700,675 parameters and a smaller network with 682,851 parameters. The larger size was chosen to limit the amount of time and GPU memory used by the model, which required one and a half weeks to fully train on our larger training set. The smaller size was chosen for use in our browser implementation because it could realistically be sent over the Internet; compressed, this network is a few hundred kilobytes. We evaluate the two sizes of models with a variety of password policies, since each policy may respond differently to size constraints, and describe the results in Section 5.1.

Transference Learning We experimented with a specialized method of training neural networks that takes advantage of *transference learning*, in which different parts of a neural network learn to recognize different phenomena during training [97]. One of the key problems with targeting non-traditional password policies is that there is little training data. For example, in our larger training set, there are 105 million passwords, but only 2.6 million satisfy a password policy that requires a minimum of 16 characters. The sparsity of training samples limits guessing approaches’ effectiveness against such non-traditional policies. However, if trained on all passwords, the learned model is non-optimal because it generates passwords that are not accurate for our target policy even if one ignores passwords that do not satisfy the policy. Transference learning lets us train a model on all passwords, yet tailor its guessing to only longer passwords.

When using transference learning, the model is first trained on all passwords in the training set. Then, the lower layers of the model are frozen. Finally, the model is retrained only on passwords in the training set that fit

the policy. The intuition is that the lower layers in the model learn low-level features about the data (e.g., that ‘a’ is a vowel), and the higher layers learn higher-level features about the data (e.g., that vowels often follow consonants). Similarly, the lower layers in the model may develop the ability to count the number of characters in a password, while the higher level layers may recognize that passwords are typically eight characters long. By fine-tuning the higher-level parameters, we can leverage what the model learned about all passwords and retarget it to a policy for which training data is sparse.

Training Data We experimented with different sets of training data; we describe experiments with two sets of passwords in Sections 4.1 and 5.2, and also with including natural language in training data in Section 5.1. For machine-learning algorithms in general, more training data is better, but only if the training data is a close match for the passwords we test on.

3.3 Client-Side Models

Deploying client-side (e.g., browser-based) password-strength-measuring tools presents severe challenges. To minimize the latency experienced by users, these tools should execute quickly and transfer as little data as possible over the network. Advanced guessing tools (e.g., PCFG, Markov models, and tools like JtR and Hashcat) run on massively parallel servers and require on the order of hundreds of megabytes or gigabytes of disk space. Typically, these models also take hours or days to return results of strength-metric tests, even with recent advances in efficient calculation [34], which is unsuitable for real-time feedback. In contrast, by combining a number of optimizations with the use of neural networks, we can build accurate password-strength-measuring tools that are sufficiently fast for real-time feedback and small enough to be included in a web page.

3.3.1 Optimizing for Model Size

To deploy our prototype implementation in a browser, we developed methods for succinctly encoding it. We leveraged techniques from graphics for encoding 3D models for browser-based games and visualizations [29]. Our encoding pipeline contains four different steps: weight quantization, fixed-point encoding, ZigZag encoding, and lossless compression. Our overall strategy is to send fewer bits and leverage existing lossless compression methods that are natively supported by browser implementations, such as gzip compression [41]. We describe the effect that each step in the pipeline has on compression in Section 5.3. We also describe encoding a short wordlist of passwords in Bloom filters.

Weight Quantization First, we quantized the weights of the neural network to represent them with fewer digits. Rather than sending all digits of the 32-bit floating-point numbers that describe weights, we only send the most significant digits. Weight quantization is routinely used for decreasing model size, but can increase error [68]. We show the effect of quantization on error rates in Section 5.3. We experimentally find that quantizing weights up to three decimal digits leads to minimal error.

Fixed-point Encoding Second, instead of representing weights using floating-point encoding, we used fixed-point encoding. Due to the weight-quantization step, many of the weight values are quantized to the same values. Fixed-point encoding allows us to more succinctly describe the quantized values using unsigned integers rather than floating point numbers on the wire: one could internally represent a quantized weight between -5.0 and 5.0 with a minimum precision of 0.005, as between -1000 and 1000 with a precision of 1. Avoiding the floating-point value would save four bytes. While lossless compression like gzip partially reduces the need for fixed-point encoding, we found that such scaling still provides an improvement in practice.

ZigZag Encoding Third, negative values are generally more expensive to send on the wire. To avoid sending negative values, we use ZigZag encoding [8]. In ZigZag encoding, signed values are encoded by using the last bit as the sign bit. So, the value of 0 is encoded as 0, but the value of -1 is encoded as 1, 1 is encoded as 2, -2 is encoded as 3, and so on.

Lossless Compression We use regular gzip or deflate encoding as the final stage of the compression pipeline. Both gzip and deflate produce similar results in terms of model size and both are widely supported natively by browsers and servers. We did not consider other compression tools, like LZMA, because their native support by browsers is not as widespread, even though they typically result in slightly smaller models.

Bloom Filter Word List To increase the success of client-side guessing, we also store a word list of frequently guessed passwords. As in previous work [89], we found that for some types of password-cracking methods, prepending training passwords improves guessing effectiveness. We stored the first two million most frequently occurring passwords in our training set in a series of compressed Bloom filters [69].

Because Bloom filters cannot map passwords to the number of guesses required to crack, and only compute

existence in a set, we use multiple Bloom filters in different groups: in one Bloom filter, we include passwords that require fewer than 10 guesses; in another, all passwords that require fewer than 100 guesses; and so on. On the client, a password is looked up in each filter and assigned a guess number corresponding to the filter with the smallest set of passwords. This allows us to roughly approximate the guess number of a password without increasing the error bounds of the Bloom filter. To drastically decrease the number of bits required to encode these Bloom filters, we only send passwords that meet the requirements of the policy and would have neural-network-computed guess numbers more than three orders of magnitude different from their actual guess numbers. We limited this word list to be about 150KB after compression in order to limit the size of our total model. We found that significantly more space would be needed to substantially improve guessing success.

3.3.2 Optimizing for Latency

We rely on precomputation and caching to make our prototype sufficiently fast for real-time feedback. Our target latency is near 100 ms because that is the threshold below which updates appear instantaneous [72].

Precomputation We precompute guess numbers instead of calculating guess numbers on demand because all methods of computing guess numbers on demand are too slow to give real-time feedback. For example, even with recent advances in calculation efficiency [34], our fastest executing model, the Markov model, requires over an hour to estimate guess numbers of our test set passwords, with other methods taking days. Precomputation decreases the latency of converting a password probability to a guess number: it becomes a quick lookup in a table on the client.

The drawback of this type of precomputation is that guess numbers become inexact due to the quantization of the probability-to-guess-number mapping. We experimentally measure (see Section 5.3) the accuracy of our estimates, finding the effect on accuracy to be low. For the purpose of password-strength estimation, we believe the drawback to be negligible, in part because results are typically presented to users in more heavily quantized form. For instance, users may be told their password is “weak” or “strong.” In addition, the inaccuracies introduced by precomputation can be tuned to result in safe errors, in that any individual password’s guess number may be an underestimate, but not an overestimate.

Caching Intermediate Results We also cache results from intermediate computations. Calculating the probability of a 10-character password requires 11 full computa-

tions of the neural network, one for each character and one for the end symbol. By caching probabilities of each substring, we significantly speed up the common case in which a candidate password changes by having a character added to or deleted from its end. We experimentally show the benefits of caching in Section 5.3.

Multiple Threads On the client side, we run the neural network computation in a separate thread from the user interface for better responsiveness of the user interface.

3.4 Implementation

We build our server-side implementation on the Keras library [28] and the client-side implementation on the neocortex browser implementation [5] of neural networks. We use the Theano back-end library for Keras, which trains neural networks faster by using a GPU rather than a CPU [17, 18]. Our implementation trains networks and guesses passwords in the Python programming language. Guess number calculation in the browser is performed in JavaScript. Our models typically used three long short-term memory (LSTM) recurrent layers and two densely connected layers for a total of five layers. On the client side, we use the WebWorker browser API to run neural network computations in their own thread [10].

For some applications, such as in a password meter, it is desirable to conservatively estimate password strength. Although we also want to minimize errors overall, on the client we prefer to underestimate a password’s resistance to guessing, rather than overestimate it. To get a stricter underestimate of guess numbers on our client-side implementation, we compute the guess number without respect to capitalization. We find in practice that our model is able to calculate a stricter underestimate this way, without overestimating many passwords’ strength. We don’t do this for the server-side models because those models are used to generate candidate password guesses, rather than estimating a guess number. After computing guess numbers, we apply to them a constant scaling factor, which acts as a security parameter, to make the model more conservative at the cost of making more errors. We discuss this tradeoff more in Section 5.3.

4 Testing Methodology

To evaluate our implementation of neural networks, we compare it to multiple other password cracking methods, including PCFGs, Markov models, JtR, and Hashcat. Our primary metric for guessing accuracy is the guessability of our test set of human-created passwords. The guessability of an individual password is measured by how many guesses a guesser would take to crack a

password. We experiment with two sets of training data and with five sets of test data. For each set of test data, we compute the percentage of passwords that would be cracked after a particular number of guesses. More accurate guessing methods correctly guess a higher percentage of passwords in our test set.

For probabilistic methods—PCFG, Markov models, and neural networks—we use recent work to efficiently compute guess numbers using Monte Carlo methods [34]. For Monte Carlo simulations, we generate and compute probabilities for at least one million random passwords to provide accurate estimates. While the exact error of this technique depends heavily on each method, guess number, and individual password, typically we observed 95% confidence intervals of less than 10% of the value of the guess-number estimate; passwords for which the error exceeded 10% tended to be guessed only after more than 10^{18} guesses. For all Monte Carlo simulations, we model up to 10^{25} guesses for completeness. This is likely an overestimate of the number of guesses that even a well-resourced attacker could be able to or would be incentivized to make against one password.

To calculate guessability of passwords using mangling-rule-based methods—JtR and Hashcat—we enumerate all guesses that these methods make. This provides exact guess numbers, but fewer guesses than we simulate with other methods. Across our different test sets, the mangling-rule-based methods make between about 10^{13} and 10^{15} guesses.

4.1 Training Data

To train our algorithms, we used a mixture of leaked and cracked password sets. We believe this is ethical because these password sets are already publicly available and we cause no additional harm with their use.

We explore two different sets of training data. We term the first set the Password Guessability Service (*PGS*) training set, used by prior work [89]. It contains the Rockyou [90] and Yahoo! [43] leaked password sets. For guessing methods that use natural language, it also includes the web2 list [11], Google web corpus [47], and an inflection dictionary [78]. This set totals 33 million passwords and 5.9 million natural-language words.

The second set (the *PGS++* training set) augments the *PGS* training set with additional leaked and cracked password sets [1, 2, 3, 6, 7, 9, 12, 13, 14, 15, 16, 20, 23, 25, 42, 43, 55, 56, 57, 62, 63, 67, 75, 77, 85, 90]. For methods that use natural language, we include the same natural-language sources as the *PGS* set. This set totals 105 million passwords and 5.9 million natural-language words.

4.2 Testing Data

For our testing data we used passwords collected from Mechanical Turk (MTurk) in the context of prior research studies, as well as a set sampled from the leak of plaintext passwords from 000webhost [40]. In addition to a common policy requiring only eight characters, we study three less common password policies shown to be more resistant to guessing [66, 80]: 4class8, 3class12, and 1class16, all described below. We chose the MTurk sets to get passwords created under more password policies than were represented in leaked data. Passwords generated using MTurk have been found to be similar to real-world, high-value passwords [38, 66]. Nonetheless, we chose the 000webhost leak to additionally compare our results to real passwords from a recently leaked password set. In summary, we used five testing datasets:

- **1class8:** 3,062 passwords longer than eight characters collected for a research study [59]
- **1class16:** 2,054 passwords longer than sixteen characters collected for a research study [59]
- **3class12:** 990 passwords that must contain at least three character classes (uppercase letters, lowercase letters, symbols, digits) and be at least twelve characters long collected for a research study [80]
- **4class8:** 2,997 passwords that must contain all four character classes and be at least eight characters long collected for a research study [66]
- **webhost:** 30,000 passwords randomly sampled from among passwords containing at least eight characters in the 000webhost leak [40]

4.3 Guessing Configuration

PCFG We used a version of PCFG with terminal smoothing and hybrid structures [60], and included natural-language dictionaries in the training data, weighted for each word to count as one tenth of a password. We also separated training for structures and terminals, and trained structures only on passwords that conform to the target policy. This method does not generate passwords that do not match the target policy.

For PCFG, Monte Carlo methods are not able to estimate unique guess numbers for passwords that have the same probability. This phenomenon manifests in the Monte Carlo graphs with jagged edges, where many different passwords are assigned the same guess number (e.g., in Figure 5c before 10^{23}). We assume that an optimal attacker could order these guesses in any order, since they all have the same likelihood according to the model. Hence, we assign the lowest guess number to all of these guesses. This is a strict overestimate of PCFG’s guessing effectiveness, but in practice does not change the results.

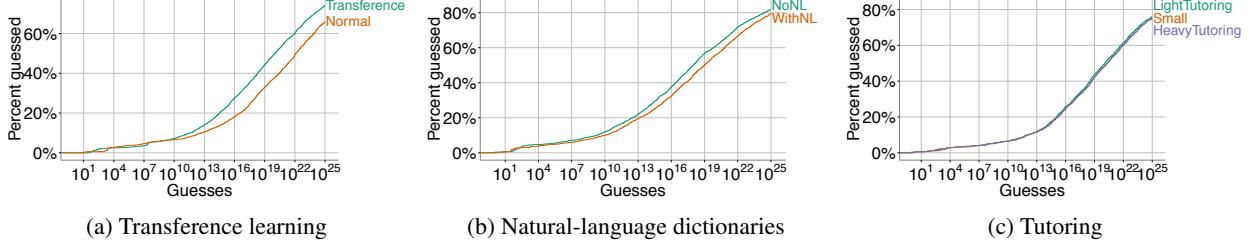


Figure 2: **Alternative training methods for neural networks.** The x -axes represent the number of guesses in log scale. The y -axes show the corresponding percentage of 1class16 passwords guessed. In (b), *WithNL* is a neural network trained with natural-language dictionaries, and *NoNL* is a neural network trained without natural-language dictionaries.

Markov Models We trained 4-, 5-, and 6-gram models. Prior work found the 6-gram models and additive smoothing of 0.01 to be an effective configuration for most password sets [65]. Our results agree, and we use the 6-gram model with additive smoothing in our tests. We discard guesses that do not match the target policy.

Mangling Wordlist Methods We compute guess numbers using the popular cracking tools Hashcat and John the Ripper (JtR). For Hashcat, we use the best64 and gen2 rule sets that are included with the software [83]. For JtR, we use the SpiderLabs mangling rules [86]. We chose these sets of rules because prior work found them effective in guessing general-purpose passwords [89]. To create the input for each tool, we uniques and sorted the respective training set by descending frequency. For JtR, we remove guesses that do not match the target policy. For Hashcat, however, we do not do so because Hashcat’s GPU implementation can suffer a significant performance penalty. We believe that this models a real-world scenario where this penalty would also be inflicted.

5 Evaluation

We performed a series of experiments to tune the training of our neural networks and compare them to existing guessing methods. In Section 5.1, we describe experiments to optimize the guessing effectiveness of neural networks by using different training methods. These experiments were chosen primarily to guide our decisions about model parameters and training along the design space we describe in Section 3.2, including training methods, model size, training data, and network architecture. In Section 5.2, we compare the effectiveness of the neural network’s guessing to other guessing algorithms. Finally, in Section 5.3, we describe our browser implementation’s effectiveness, speed, and size, and we compare it to other browser password-measuring tools.

5.1 Training Neural Networks

We conducted experiments exploring how to tune neural network training, including modifying the network size, using sub-word models, including natural-language dictionaries in training, and exploring alternative architectures. We do not claim that these experiments are a complete exploration of the space. Indeed, improving neural networks is an active area of research.

Transference Learning We find that the transference learning training, described in Section 3.2, improves guessing effectiveness. Figure 2a shows in log scale the effect of transference learning. For example, at 10^{15} guesses, 22% of the test set has been guessed with transference learning, as opposed to 15% without transference learning. Using a 16 MB network, we performed this experiment on our 1class16 passwords because they are particularly different from the majority of our training set. Here, transference learning improves password guessing mostly at higher guess numbers.

Including Natural-Language Dictionaries We experimented with including natural-language dictionaries in the neural network training data, hypothesizing that doing so would improve guessing effectiveness. We performed this experiment with 1class16 passwords because they are particularly likely to benefit from training on natural-language dictionaries [91]. Networks both with and without natural language data were trained using the transference learning method on long passwords. Natural language was included with the primary batch of training data. Figure 2b shows that, contrary to our hypotheses, training on natural language decreases the neural network’s guessing effectiveness. We believe neural networks do not benefit from natural language, in contrast to other methods like PCFG, because this method of training does not differentiate between natural-language dictionaries and password training. However, training data could be enhanced with natural language in other ways, perhaps yielding better results.

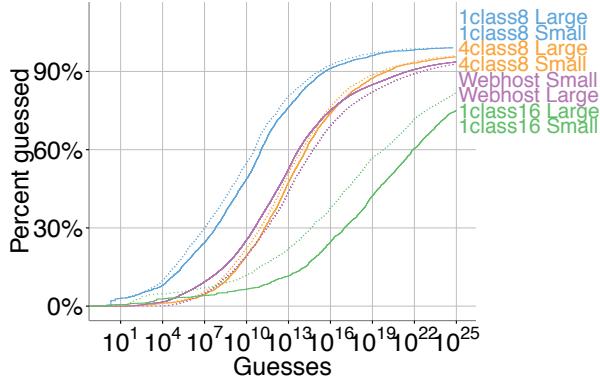


Figure 3: **Neural network size and password guessability.** Dotted lines are large networks; solid lines are small networks.

Password Tokenization We find that using hybrid, sub-word level password models does not significantly increase guessing performance at low guess numbers. Hybrid models may represent the same word in multiple different ways. For example, the model may capture a word as one token, ‘pass’, or as the letters ‘p’, ‘a’, ‘s’, ‘s’. Because Monte Carlo simulations assume that passwords are uniquely represented, instead of using Monte Carlo methods to estimate guess numbers, we calculated guess numbers by enumerating the most probable 10^7 guesses. However, at this low number of guesses, we show this tokenization has only a minor effect, as shown in Figure 4b. We conducted this experiment on long passwords because we believed that they would benefit most from tokenization. This experiment shows that there may be an early benefit, but otherwise the models learn similarly. We consider this result to be exploratory both due to our low guessing cutoff and because other options for tuning the tokenization could produce better results.

Model Size We find that, for at least some password sets, neural network models can be orders of magnitude smaller than other models with little effect on guessing effectiveness. We tested how the following two model sizes impact guessing effectiveness: a large model with 1,000 LSTM cells or 15,700,675 parameters that uses 60 MB, and a small model with 200 LSTM cells or 682,851 parameters that takes 2.7 MB.

The results of these experiments are shown in Figure 3. For 1class8 and 4class8 policies, the effect of decreasing model size is minor but noticeable. However, for 1class16 passwords, the effect is more dramatic. We attribute differences between the longer and shorter policies with respect to model size to fundamental differences in password composition between those policies. Long passwords are more similar to English language phrases, and modeling them may require more parameters, and hence larger networks, than modeling shorter

passwords. The webhost test set is the only set for which the larger model performed worse. We believe that this is due to the lack of suitability of the particular training data we used for this model. We discuss the differences in training data more in Section 5.2.

Tutored Networks To improve the effectiveness of our small model at guessing long passwords, we attempted to tutor our small neural network with randomly generated passwords from the larger network. While this had a mild positive effect with light tutoring, at a roughly one to two ratio of random data to real data, the effect does not seem to scale to heavier tutoring. Figure 2c shows minimal difference in guessing accuracy when tutoring is used, and regardless of whether it is light or heavy.

Backwards vs. Forwards Training As described in Section 3.2, processing input backwards rather than forwards can be more effective in some applications of neural networks [48]. We experiment with guessing passwords backwards, forwards, and using a hybrid approach where half of the network examines passwords forwards and the other half backwards. We observed only marginal differences overall. At the point of greatest difference, near 10^9 guesses, the hybrid approach guessed 17.2% of the test set, backwards guessed 16.4% of the test set and forwards guessed 15.1% of the test set. Figure 4a shows the result of this experiment. Since the hybrid approach increases the amount of time required to train with only small improvement in accuracy, for other experiments we use backwards training.

Recurrent Architectures We experimented with two different types of recurrent neural-network architectures: long short-term memory (LSTM) models [54] and a refinement on LSTM models [58]. We found that this choice had little effect on the overall output of the network, with the refined LSTM model being slightly more accurate, as shown in Figure 4c.

5.2 Guessing Effectiveness

Compared to other individual password-guessing methods, we find that neural networks are better at guessing passwords at a higher number of guesses and when targeting more complex or longer password policies, like our 4class8, 1class16, and 3class12 data sets. For example, as shown in Figure 5b, neural networks guessed 70% of 4class8 passwords by 10^{15} guesses, while the next best performing guessing method guesses 57%.

Models differ in how effectively they guess specific passwords. *MinGuess*, shown in Figure 5, represents an idealized guessing approach in which a password is considered guessed as soon as it is guessed by any of our

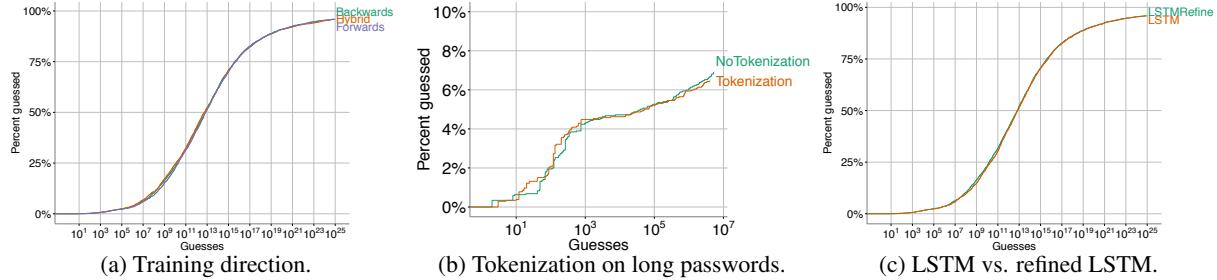


Figure 4: **Additional tuning experiments.** Our LSTM experiments tested on complex passwords with 16M parameters. We found very little difference in performance. Our experiments on tokenization examined long passwords. Our experiments on training direction involved training backwards, forwards, and both backwards and forwards with 16M parameters on complex passwords.

guessing approaches, including neural networks, Markov models, PCFG, JtR, and Hashcat. That MinGuess outperforms neural networks suggests that using multiple guessing methods should still be preferred to using any single guessing method for accurate strength estimation, despite the fact that neural networks generally outperform other models individually.

For all the password sets we tested, neural networks outperformed other models beginning at around 10^{10} guesses, and matched or beat the other most effective methods before that point. Figures 5–6 show the performance of the different guessing methods trained with the PGS data set, and Figures 7–8 show the same guessing methods trained with the PGS++ data set. Both data sets are described in more detail in Section 4.1. In this section, we used our large, 15.7 million parameter neural network, trained with transference learning on two training sets. While performance varies across guessing method and training set, in general we find that the neural networks’ performance at high guess numbers and across policies holds for both sets of training data with one exception, discussed below. Because these results hold for multiple training and test sets, we hypothesize that neural networks would also perform well in guessing passwords created under many policies that we did not test.

In the webhost test set using the PGS++ training data, neural networks performed worse than other methods. For webhost, all guessing methods using the PGS++ data set were less effective than the PGS data set, though some methods, such as PCFG, were only slightly affected. Because all methods perform worse, and because, when using the PGS training data, neural networks do better than other methods—similar to other test sets—we believe that the PGS++ training data is particularly ineffective for this test set. As Figure 3 shows, this is the only data set where a smaller neural network performs significantly better than the larger neural network, which suggests that the larger neural network model is fitting itself more strictly to low-quality data, which limits the larger network’s ability to generalize.

Qualitatively, the types of passwords that our implementation of neural networks guessed before other methods were novel passwords that were dissimilar to passwords in the training set. The types of passwords that our implementation of neural networks were late to guess but that were easily guessable by other methods often were similar to words in the natural-language dictionaries, or were low-frequency occurrences in the training data.

Resource Requirements In general, PCFGs require the most disk, memory, and computational resources. Our PCFG implementation stored its grammar in 4.7GB of disk space. Markov models are the second largest of our implementations, requiring 1.1GB of disk space. Hashcat and JtR do not require large amounts of space for their rules, but do require storing the entire training set, which is 756MB. In contrast, our server-side neural network requires only 60MB of disk space. While 60MB is still larger than what could effectively be transferred to a client without compression, it is a substantial improvement over the other models.

5.3 Browser Implementation

While effective models can fit into 60MB, this is still too large for real-time password feedback in the browser. In this section, we evaluate our techniques for compressing neural network models, discussed in Section 3.3, by comparing the guessing effectiveness of the compressed models to all server-side models—our large neural network, PCFG, Markov models, JtR, and Hashcat.

Model Encoding Our primary size metric is the gziped model size. Our compression stages use the JSON format because of its native support in JavaScript platforms. We explored using the MsgPack binary format [4], but found that after gzip compression, there was no benefit for encoding size and minor drawbacks for decoding speed. The effects of different pipeline stages on compression are shown in Table 1.

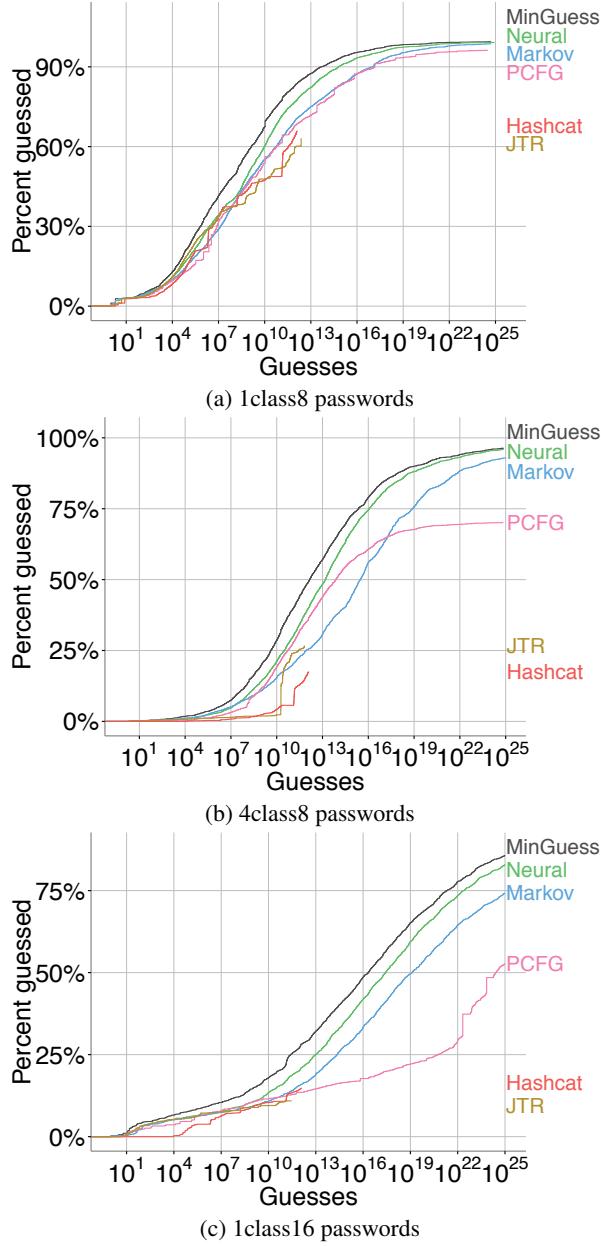


Figure 5: **Guessability of our password sets for different guessing methods using the PGS data set.** *MinGuess* stands for the minimum number of guesses for any approach. Y-axes are differently scaled to best show comparative performance.

Weight and Probability Curve Quantization Because current methods of calculating guess numbers from probabilities are too slow, taking hours or days to return results, we precompute a mapping from password probability to guess number and send the mapping to the client, as described in Section 3.3.2. Such a mapping can be efficiently encoded by quantizing the probability-to-guess-number curve. Quantizing the curve incurs safe errors—i.e., we underestimate the strength of passwords.

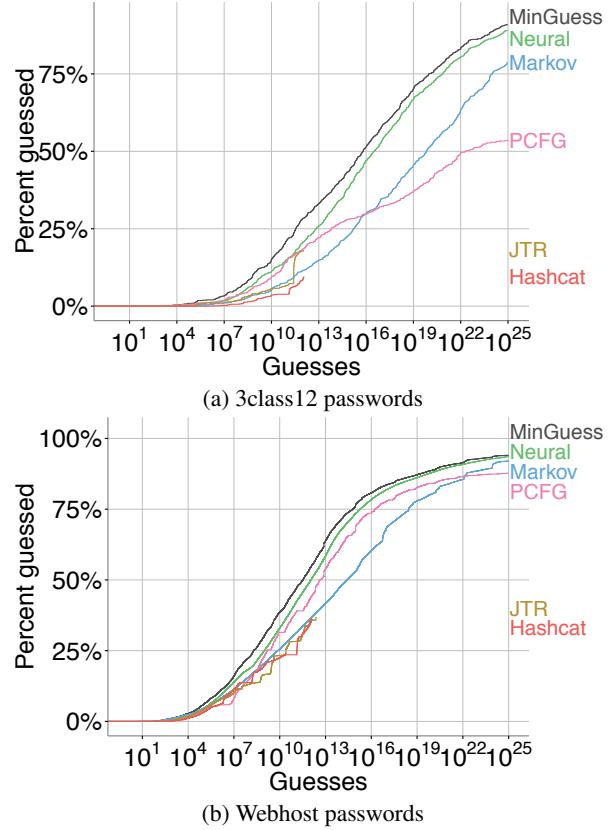


Figure 6: **Guessability of our password sets for different guessing methods using the PGS data set (continued).**

We also quantize the model’s parameters in the browser implementation to further decrease the size of the model. Both weight and curve quantization are lossy operations, whose effect on guessing we show in Figure 9. Curve quantization manifests in a saw-tooth shape to the guessing curve, but the overall shape of the guessing curve is largely unchanged.

Evaluating Feedback Speed Despite the large amount of computation necessary for computing a password’s guessability, our prototype implementation is efficient enough to give real-time user feedback. In general, feedback quicker than 100 ms is perceived as instantaneous [72]; hence, this was our benchmark. We performed two tests to measure the speed of calculating guess numbers: the first measures the time to produce guess numbers with a semi-cached password; the second computes the total time per password. The semi-cached test measures the time to compute a guess number when adding a character to the end of a password. We believe this is representative of what a user would experience in practice because a user typically creates a password by typing it in character by character.

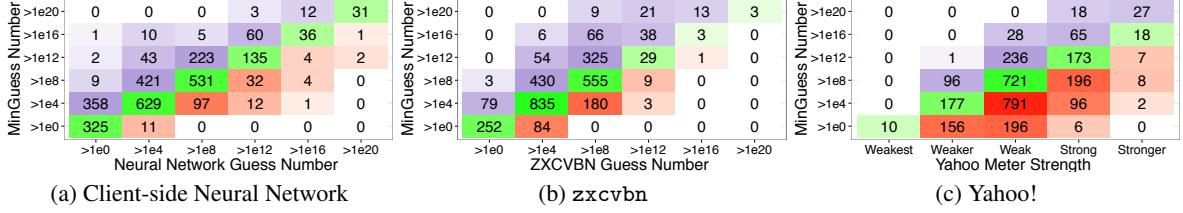


Figure 10: **Client-side guess numbers compared to the minimum guess number of all server-side methods.** The number in the bin represents the number of passwords in that bin. For example, neural networks rated 358 passwords as being guessed with between 10^0 and 10^4 guesses, while server-side approaches rate them as taking between 10^4 and 10^8 guesses. The test passwords are our 1class8 set. The Yahoo! meter does not provide guess numbers and, as such, has a different x -axis. Overestimates of strength are shown in shades of red, underestimates in shades of purple, and accurate estimates in shades of green. Color intensity rises with the number of passwords in a bin.

Pipeline stage	Size	gzip-ed Size
Original JSON format	6.9M	2.4M
Quantization	4.1M	716K
Fixed point	3.1M	668K
ZigZag encoding	3.0M	664K
Removing spaces	2.4M	640K

Table 1: **The effect of different pipeline stages on model size.** This table shows the small model that targets the 1class8 password policy, with 682,851 parameters. Each stage includes the previous stage, e.g., the fixed-point stage includes the quantization stage. We use gzip at the highest compression level.

		Total	Unsafe
1class8	Neural Network	1311	164
	zxcvbn	1331	270
	Yahoo!	1900	984
4class8	Neural Network	1826	115
	zxcvbn	1853	231
	Yahoo!	1328	647

Table 2: **The number of total and unsafe misclassifications for different client-side meters.** Because the Yahoo! meter provides different binning, we pre-process its output for fairer comparison, as described in Section 5.3.

We perform both tests on a laptop running OSX with a 2.7 GHz i7 processor and using the Chrome web browser (version 48). We randomly selected a subset of 500 passwords from our 1class8 training set for these tests. In the semi-cached test, the average time to compute a guess number is 17 ms (stdev: 4 ms); in the full-password test, the average time is 124 ms (stdev: 48 ms). However, both the semi-cached test and the uncached test perform fast enough to give quick feedback to users.

Comparison to Other Password Meters We compared the accuracy of our client-side neural network implementation to other client-side password-strength es-

timators. Approximations of password strength can be under- or overestimates. We call overestimates of password strength unsafe errors, since they represent passwords as harder to guess than they actually are. We show that our meter can more precisely measure passwords’ resistance to guessing with up to half as many unsafe errors as existing client-side models, which are based on heuristics. Our ground truth for this section is the idealized MinGuess method, described in Section 5.2.

Prior work found nearly all proactive password-strength estimators to be inconsistent and to poorly estimate passwords’ resistance to guessing [33]. The most promising estimator was Dropbox’s zxcvbn meter [94, 95], which relies on hand-crafted heuristics, statistical methods, and plaintext dictionaries as training data to estimate guess numbers. Notably, these plaintext dictionaries are not the same as those used for our training data, limiting our ability to fully generalize from these comparisons. Exploring other ways of configuring zxcvbn is beyond the scope of this evaluation. We compare our results to both zxcvbn and the Yahoo! meter, which is an example of using far less sophisticated heuristics to estimate password strength.

The Yahoo! meter does not produce guess numbers but bins passwords as weakest, weaker, weak, strong, and stronger. We ignore the semantic values of the bin names, and examine the accuracy with which the meter classified passwords with different guess numbers (as computed by the MinGuess of all guessing methods) into the five bins. To compare the Yahoo! meter to our minimum guess number (Table 2), we take the median actual guess number of each bin (e.g., the “weaker” bin) and then map the minimum guess number for each password to the bin that it is closest to on a log scale. For example, in the Yahoo! meter, the guess number of $5.4 \cdot 10^4$ is the median of the “weaker” bin; any password closer to $5.4 \cdot 10^4$ than to the medians of other bins on a log scale we consider as belonging in the “weaker” bin. We intend for this to be an overestimate of the accuracy of

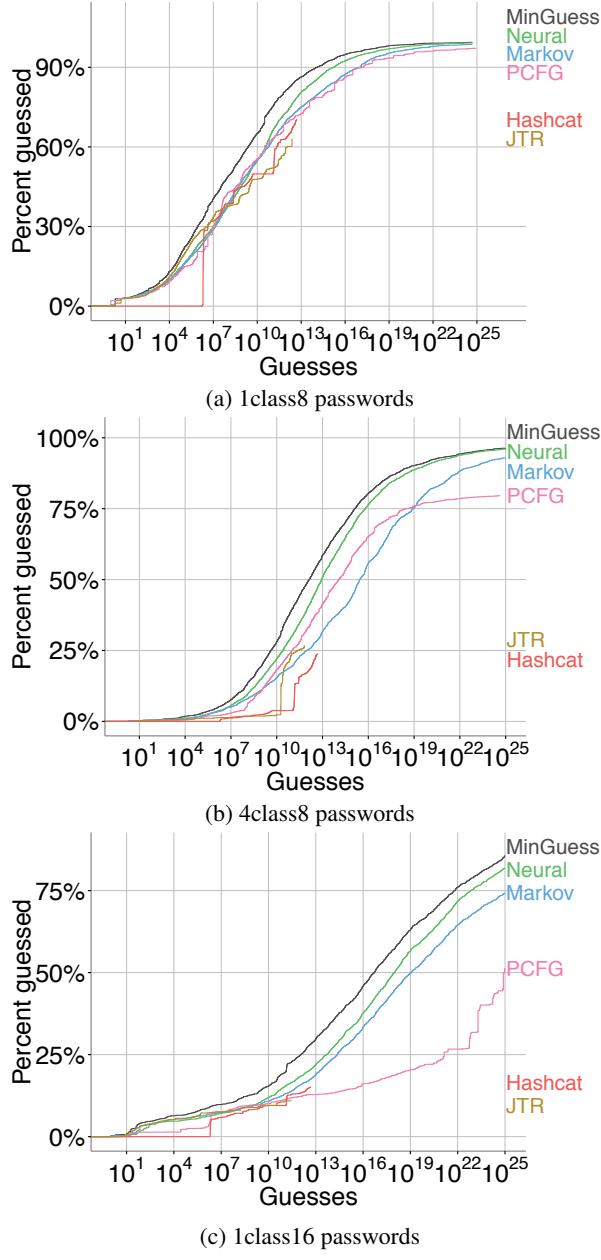


Figure 7: **Guessability of our password sets for different guessing methods using the PGS++ data set.** *MinGuess* stands for the minimum number of guesses for any approach.

the Yahoo! meter. Nonetheless, both our work and prior work [33] find the Yahoo! meter to be less accurate than other approaches, including the `zxcvbn` meter.

We find that our client-side neural network approach is more accurate than the other approaches we test, with up to two times fewer unsafe errors and comparable safe errors, as shown in Figure 10 and Table 2. Here, we used our neural network meter implementation with the tuning described in Section 3.4. We performed the 1class8

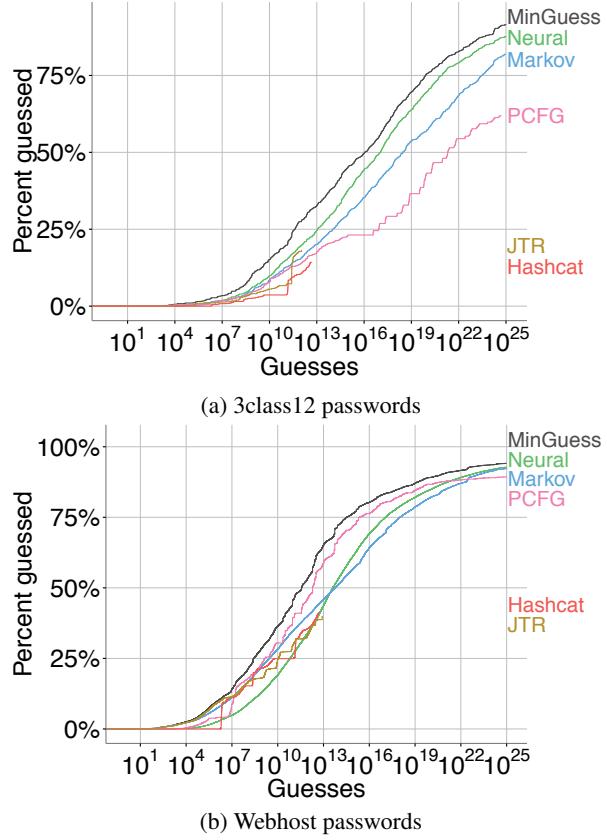


Figure 8: **Guessability of our password sets for different guessing methods using the PGS++ data set (continued).**

test with the client-side Bloom filter, described in Section 3.3.1, while the 4class8 test did not use the Bloom filter because it did not significantly impact accuracy. Both tests scale the network output down by a factor of 300 and ignore case to give more conservative guess numbers. We chose the scaling factor to tune the network to make about as many safe errors as `zxcvbn`. In addition, we find that, compared to our neural network implementation, the `zxcvbn` meter’s errors are often at very low guess numbers, which can be particularly unsafe. For example, for the 10,000 most likely passwords, `zxcvbn` makes 84 unsafe errors, while our neural network only makes 11 unsafe errors.

Besides being more accurate, we believe the neural network approach is easier to apply to other password policies. The best existing meter, `zxcvbn`, is hand-crafted to target one specific password policy. On the other hand, neural networks enable easy retargeting to other policies simply by retraining.

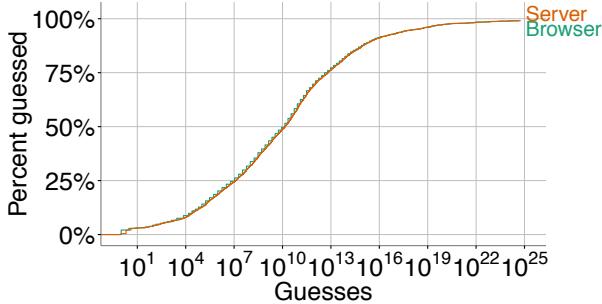


Figure 9: **Compressed browser neural network with weight and curve quantization compared an unquantized network.** *Browser* is our browser network with weight and curve quantization. *Server* is the same small neural network without weight and curve quantization.

6 Conclusion

This paper describes how to use neural networks to model human-chosen passwords and measure password strength. We show how to build and train neural networks that outperform state-of-the-art password-guessing approaches in efficiency and effectiveness, particularly for non-traditional password policies and at guess numbers above 10^{10} . We also demonstrate how to compress neural network password models so that they can be downloaded as part of a web page. This makes it possible to build client-side password meters that provide a good measure of password strength.

Tuning neural networks for password guessing and developing accurate client-side password-strength metrics both remain fertile research grounds. Prior work has used neural networks to learn the output of a larger ensemble of models [24] and obtained better results than our network tutoring (Section 5.1). Other work achieves higher compression ratios for neural networks than we do by using matrix factorization or specialized training methods [51, 96]. Further experiments on leveraging natural language, tokenized models, or other neural-networks architectures might allow passwords to be guessed more effectively. While we measured client-side strength metrics based on guessing effectiveness, a remaining challenge is giving user-interpretable advice to improve passwords during password creation.

7 Acknowledgements

We would like to thank Mahmood Sharif for participating in discussions about neural networks and Dan Wheeler for his feedback. This work was supported in part by gifts from the PNC Center for Financial Services Innovation, Microsoft Research, and John & Claire Bertucci.

References

- [1] CSDN password leak. http://thepasswordproject.com/leaked_password_lists_and_dictionaries.
- [2] Faith writer leak. https://wiki.skullsecurity.org/Passwords#Leaked_passwords.
- [3] Hak5 leak. https://wiki.skullsecurity.org/Passwords#Leaked_passwords.
- [4] Msgpack: It's like JSON but fast and small. <http://msgpack.org/index.html>.
- [5] Neocortex Github repository. <https://github.com/scienceai/neocortex>.
- [6] Perl monks password leak. <http://news.softpedia.com/news/PerlMonks-ZF0-Hack-Has-Wider-Implications-118225.shtml>.
- [7] Phpbp password leak. <https://wiki.skullsecurity.org/Passwords>.
- [8] Protocol buffer encoding. <https://developers.google.com/protocol-buffers/docs/encoding>.
- [9] Stratfor leak. http://thepasswordproject.com/leaked_password_lists_and_dictionaries.
- [10] Using Web workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. Accessed:Feb 2016.
- [11] The “web2” file of English words. <http://www.bee-man.us/computer/grep/grep.htm#web2>, 2004.
- [12] Password leaks: Elitehacker. <https://wiki.skullsecurity.org/Passwords>, 2009.
- [13] Password leaks: Alypaa. <https://wiki.skullsecurity.org/Passwords>, 2010.
- [14] Specialforces.com password leak. <http://www.databreaches.net/update-specialforces-com-hackers-acquired-8000-credit-card-numbers/>, 2011.
- [15] YouPorn password leak, 2012. http://thepasswordproject.com/leaked_password_lists_and_dictionaries.
- [16] WOM Vegas password leak, 2013. <https://www.hackread.com/wom-vegas-breached-10000-user-accounts-leaked-by-darkweb-goons/>.
- [17] BASTIEN, F., LAMBLIN, P., PASCANU, R., BERGSTRA, J., GOODFELLOW, I. J., BERGERON, A., BOUCHARD, N., AND BENGIO, Y. Theano: New features and speed improvements. In *Proc. NIPS 2012 Deep Learning workshop* (2012).
- [18] BERGSTRA, J., BREULEUX, O., BASTIEN, F., LAMBLIN, P., PASCANU, R., DESJARDINS, G., TURIAN, J., WARDE-FARLEY, D., AND BENGIO, Y. Theano: A CPU and GPU math expression compiler. In *Proc. SciPy* (2010).
- [19] BISHOP, M., AND KLEIN, D. V. Improving system security via proactive password checking. *Computers & Security* 14, 3 (1995), 233–249.
- [20] BONNEAU, J. The Gawker hack: How a million passwords were lost. *Light Blue Touchpaper* Blog, December 2010. <http://www.lightbluetouchpaper.org/2010/12/15/the-gawker-hack-how-a-million-passwords-were-lost/>.
- [21] BONNEAU, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proc. IEEE Symp. Security & Privacy* (2012).
- [22] BONNEAU, J. Statistical metrics for individual password strength. In *Proc. WPS* (2012).

- [23] BRODKIN, J. 10 (or so) of the worst passwords exposed by the LinkedIn hack. *Ars Technica*, June 6, 2012. <http://arstechnica.com/security/2012/06/10-or-so-of-the-worst-passwords-exposed-by-the-linkedin-hack/>.
- [24] BUCILUĀ, C., CARUANA, R., AND NICULESCU-MIZIL, A. Model compression. In *Proc. KDD* (2006).
- [25] BURNETT, M. Xato password set. <https://xato.net/>.
- [26] CASTELLUCCIA, C., DÜRMUTH, M., AND PERITO, D. Adaptive password-strength meters from Markov models. In *Proc. NDSS* (2012).
- [27] CHANG, J. M. Passwords and email addresses leaked in Kickstarter hack attack. *ABC News*, Feb 17, 2014. <http://abcnews.go.com/Technology/passwords-email-addresses-leaked-kickstarter-hack/story?id=22553952>.
- [28] CHOLLET, F. Keras Github repository. <https://github.com/fchollet/keras>.
- [29] CHUN, W. WebGL Models: End-to-End. In *OpenGL Insights*. 2012.
- [30] CIARAMELLA, A., D'ARCO, P., DE SANTIS, A., GALDI, C., AND TAGLIAFERRI, R. Neural network techniques for proactive password checking. *IEEE TDSC* 3, 4 (2006), 327–339.
- [31] CLERCQ, J. D. Resetting the password of the KRBTGT active directory account, 2014. <http://windowsitpro.com/security/resetting-password-krbtgt-active-directory-account>.
- [32] DAS, A., BONNEAU, J., CAESAR, M., BORISOV, N., AND WANG, X. The tangled web of password reuse. In *Proc. NDSS* (2014).
- [33] DE CARNÉ DE CARNAVALET, X., AND MANNAN, M. From very weak to very strong: Analyzing password-strength meters. In *Proc. NDSS* (2014).
- [34] DELL'AMICO, M., AND FILIPPONE, M. Monte Carlo strength evaluation: Fast and reliable password checking. In *Proc. CCS* (2015).
- [35] DELL'AMICO, M., MICHIARDI, P., AND ROUDIER, Y. Password strength: An empirical analysis. In *Proc. INFOCOM* (2010).
- [36] DUCKETT, C. Login duplication allows 20m Alibaba accounts to be attacked. *ZDNet*, February 5, 2016. <http://www.zdnet.com/article/login-duplication-allows-20m-alibaba-accounts-to-be-attacked/>.
- [37] DÜRMUTH, M., ANGELSTORE, F., CASTELLUCCIA, C., PERITO, D., AND CHAABANE, A. OMEN: Faster password guessing using an ordered markov enumerator. In *Proc. ESSoS* (2015).
- [38] FAHL, S., HARBACH, M., ACAR, Y., AND SMITH, M. On the ecological validity of a password study. In *Proc. SOUPS* (2013).
- [39] FLORÊNCIO, D., HERLEY, C., AND VAN OORSCHOT, P. C. An administrator's guide to internet password research. In *Proc. USENIX LISA* (2014).
- [40] FOX-BREWSTER, T. 13 million passwords appear to have leaked from this free web host. *Forbes*, October 28, 2015. <http://www.forbes.com/sites/thomasbrewster/2015/10/28/000webhost-database-leak/>.
- [41] GAILLY, J.-L. gzip. <http://www.gzip.org/>.
- [42] GOODIN, D. 10,000 Hotmail passwords mysteriously leaked to web. *The Register*, October 5, 2009. http://www.theregister.co.uk/2009/10/05/hotmail_passwords_leaked/.
- [43] GOODIN, D. Hackers expose 453,000 credentials allegedly taken from Yahoo service. *Ars Technica*, July 12, 2012. <http://arstechnica.com/security/2012/07/yahoo-service-hacked/>.
- [44] GOODIN, D. Anatomy of a hack: How crackers ransom passwords like “qeadzcwrsfv1331”. *Ars Technica*, May 27, 2013. <http://arstechnica.com/security/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/>.
- [45] GOODIN, D. Why LivingSocial's 50-million password breach is graver than you may think. *Ars Technica*, April 27, 2013. <http://arstechnica.com/security/2013/04/why-livingsocials-50-million-password-breach-is-graver-than-you-may-think/>.
- [46] GOODIN, D. Once seen as bulletproof, 11 million+ Ashley Madison passwords already cracked. *Ars Technica*, September 10, 2015. <http://arstechnica.com/security/2015/09/once-seen-as-bulletproof-11-million-ashley-madison-passwords-already-cracked/>.
- [47] GOOGLE. Web 1T 5-gram version 1, 2006. <http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13>.
- [48] GRAVES, A. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, 2012.
- [49] GRAVES, A. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850, 2013.
- [50] GREENBERG, A. The police tool that perverts use to steal nude pics from Apple's iCloud. *Wired*, September 2, 2014. <https://www.wired.com/2014/09/eppb-icloud/>.
- [51] HAN, S., MAO, H., AND DALLY, W. J. A deep neural network compression pipeline: Pruning, quantization, Huffman encoding. arXiv preprint arXiv:1510.00149, 2015.
- [52] HENRY, A. Five best password managers. *LifeHacker*, January 11, 2015. <http://lifehacker.com/5529133/>.
- [53] HERLEY, C., AND VAN OORSCHOT, P. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy Magazine* 10, 1 (Jan. 2012), 28–36.
- [54] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
- [55] HUNT, T. A brief Sony password analysis. <http://www.troyhunt.com/2011/06/brief-sony-password-analysis.html>, 2011.
- [56] HUYNH, T. ABC Australia hacked nearly 50,000 user credentials posted online, half cracked in 45 secs. *Techgeek*, February 27, 2013. <http://techgeek.com.au/2013/02/27/abc-australia-hacked-nearly-50000-user-credentials-posted-online/>.
- [57] JOHNSTONE, L. 9,885 user accounts leaked from Intercessors for America by Anonymous. <http://www.cyberwarnews.info/2013/07/24/9885-user-accounts-leaked-from-intercessors-for-america-by-anonymous/>, 2013.
- [58] JOZEFOWICZ, R., ZAREMBA, W., AND SUTSKEVER, I. An empirical exploration of recurrent network architectures. In *Proc. ICML* (2015).
- [59] KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND LOPEZ, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proc. IEEE Symp. Security & Privacy* (2012).
- [60] KOMANDURI, S. *Modeling the adversary to evaluate password strength with limited samples*. PhD thesis, Carnegie Mellon University, 2016.

- [61] KOMANDURI, S., SHAY, R., CRANOR, L. F., HERLEY, C., AND SCHECHTER, S. Telepathwords: Preventing weak passwords by reading users' minds. In *Proc. USENIX Security* (2014).
- [62] KREBS, B. Fraud bazaar carders.cc hacked. <http://krebsonsecurity.com/2010/05/fraud-bazaar-carders-cc-hacked/>.
- [63] LEE, M. Hackers have released what they claim are the details of over 21,000 user accounts belonging to Billabong customers. *ZDNet*, July 13, 2012. <http://www.zdnet.com/article/over-21000-plain-text-passwords-stolen-from-billabong/>.
- [64] LOWERRE, B. T. *The HARPY speech recognition system*. PhD thesis, Carnegie Mellon University, 1976.
- [65] MA, J., YANG, W., LUO, M., AND LI, N. A study of probabilistic password models. In *Proc. IEEE Symp. Security & Privacy* (2014).
- [66] MAZUREK, M. L., KOMANDURI, S., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., KELLEY, P. G., SHAY, R., AND UR, B. Measuring password guessability for an entire university. In *Proc. CCS* (2013).
- [67] MCALLISTER, N. Twitter breach leaks emails, passwords of 250,000 users. *The Register*, Feb 2, 2013.
- [68] MIKOLOV, T., SUTSKEVER, I., DEORAS, A., LE, H.-S., KOMBRINK, S., AND CERNOCKY, J. Subword language modeling with neural networks. Preprint (<http://www.fit.vutbr.cz/~imikolov/rnnlm/char.pdf>), 2012.
- [69] MITZENMACHER, M. Compressed Bloom filters. *IEEE/ACM Transactions on Networking (TON)* 10, 5 (2002), 604–612.
- [70] NARAYANAN, A., AND SHMATIKOV, V. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. CCS* (2005).
- [71] NEEF, S. Using neural networks for password cracking. Blog post. <https://0day.work/using-neural-networks-for-password-cracking/>, 2016.
- [72] NIELSEN, J., AND HACKOS, J. T. *Usability engineering*, vol. 125184069. Academic press Boston, 1993.
- [73] PERLROTH, N. Adobe hacking attack was bigger than previously thought. *The New York Times Bits Blog*, October 29, 2013. <http://bits.blogs.nytimes.com/2013/10/29/adobe-online-attack-was-bigger-than-previously-thought/>.
- [74] PESLYAK, A. John the Ripper. <http://www.openwall.com/john/>, 1996.
- [75] PROTALINSKI, E. 8.24 million Gamigo passwords leaked after hack. *ZDNet*, July 23, 2012. <http://www.zdnet.com/article/8-24-million-gamigo-passwords-leaked-after-hack/>.
- [76] RAGAN, S. Mozilla's bug tracking portal compromised, reused passwords to blame. *CSO*, September 4, 2015. <http://www.csoonline.com/article/2980758/>.
- [77] SCHNEIER, B. MySpace passwords aren't so dumb. <http://www.wired.com/politics/security/commentary/securitymatters/2006/12/72300>, 2006.
- [78] SCOWL. Spell checker oriented word lists. <http://wordlist.sourceforge.net>, 2015.
- [79] SHAY, R., BAUER, L., CHRISTIN, N., CRANOR, L. F., FORGET, A., KOMANDURI, S., MAZUREK, M. L., MELICHER, W., SEGRETI, S. M., AND UR, B. A spoonful of sugar? The impact of guidance and feedback on password-creation behavior. In *Proc. CHI* (2015).
- [80] SHAY, R., KOMANDURI, S., DURITY, A. L., HUH, P. S., MAZUREK, M. L., SEGRETI, S. M., UR, B., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Can long passwords be secure and usable? In *Proc. CHI* (2014).
- [81] SONG, D. X., WAGNER, D., AND TIAN, X. Timing analysis of keystrokes and timing attacks on SSH. In *Proc. USENIX Security Symposium* (2001).
- [82] SPIDERoAK. Zero knowledge cloud solutions. <https://spideroak.com/>, 2016.
- [83] STEUBE, J. Hashcat. <https://hashcat.net/oclhashcat/>, 2009.
- [84] SUTSKEVER, I., MARTENS, J., AND HINTON, G. E. Generating text with recurrent neural networks. In *Proc. ICML* (2011).
- [85] TRUSTWAVE. eHarmony password dump analysis, June 2012. <http://blog.spiderlabs.com/2012/06/eharmony-password-dump-analysis.html>.
- [86] TRUSTWAVE SPIDERLABS. SpiderLabs/KoreLogic-Rules. <https://github.com/SpiderLabs/KoreLogic-Rules>, 2012.
- [87] TSUKAYAMA, H. Evernote hacked; millions must change passwords. *Washington Post*, March 4, 2013. https://www.washingtonpost.com/8279306c-84c7-11e2-98a3-b3db6b9ac586_story.html.
- [88] UR, B., KELLEY, P. G., KOMANDURI, S., LEE, J., MAASS, M., MAZUREK, M., PASSARO, T., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. How does your password measure up? The effect of strength meters on password creation. In *Proc. USENIX Security* (2012).
- [89] UR, B., SEGRETI, S. M., BAUER, L., CHRISTIN, N., CRANOR, L. F., KOMANDURI, S., KURILOVA, D., MAZUREK, M. L., MELICHER, W., AND SHAY, R. Measuring real-world accuracies and biases in modeling password guessability. In *Proc. USENIX Security* (2015).
- [90] VANCE, A. If your password is 123456, just make it hackme. *New York Times*, January 20, 2010. <http://www.nytimes.com/2010/01/21/technology/21password.html>.
- [91] VERAS, R., COLLINS, C., AND THORPE, J. On the semantic patterns of passwords and their security impact. In *Proc. NDSS* (2014).
- [92] WEIR, M., AGGARWAL, S., COLLINS, M., AND STERN, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proc. CCS* (2010).
- [93] WEIR, M., AGGARWAL, S., MEDEIROS, B. D., AND GLODEK, B. Password cracking using probabilistic context-free grammars. In *Proc. IEEE Symp. Security & Privacy* (2009).
- [94] WHEELER, D. zxcvbn: Realistic password strength estimation. <https://blogs.dropbox.com/tech/2012/04/zxcvbn-realistic-password-strength-estimation/>, 2012.
- [95] WHEELER, D. L. zxcvbn: Low-budget password strength estimation. In *Proc. USENIX Security* (2016).
- [96] XUE, J., LI, J., YU, D., SELTZER, M., AND GONG, Y. Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network. In *Proc. ICASSP* (2014).
- [97] YOSINSKI, J., CLUNE, J., BENGIO, Y., AND LIPSON, H. How transferable are features in deep neural networks? In *Proc. NIPS* (2014).

An Empirical Study of Textual Key-Fingerprint Representations

Sergej Dechand

USECAP, University of Bonn

Yasemin Acar

CISPA, Saarland University

Dominik Schürmann

IBR, TU Braunschweig

Sascha Fahl

CISPA, Saarland University

Karoline Busse

USECAP, University of Bonn

Matthew Smith

USECAP, University of Bonn

Abstract

Many security protocols still rely on manual fingerprint comparisons for authentication. The most well-known and widely used key-fingerprint representation are hexadecimal strings as used in various security tools. With the introduction of end-to-end security in WhatsApp and other messengers, the discussion on how to best represent key-fingerprints for users is receiving a lot of interest.

We conduct a 1047 participant study evaluating six different textual key-fingerprint representations with regards to their performance and usability. We focus on textual fingerprints as the most robust and deployable representation.

Our findings show that the currently used hexadecimal representation is more prone to partial preimage attacks in comparison to others. Based on our findings, we make the recommendation that two alternative representations should be adopted. The highest attack detection rate and best usability perception is achieved with a sentence-based encoding. If language-based representations are not acceptable, a simple numeric approach still outperforms the hexadecimal representation.

1 Introduction

Public key cryptography is a common method for authentication in secure end-to-end communication and has been a part of the Internet throughout the last two decades [7, 11]. While security breaches have shown that systems based on centralized trusted third parties such as Certificate Authorities and Identity Based Private Key Generators are prone to targeted attacks [42], decentralized approaches such as Web of Trust and Namecoin struggle with being adopted in practice due to usability issues [7, 13, 30]. Certificate transparency systems, such as CONIKS and others [24, 39, 27], aim to solve a subset of these issues by providing an auditable directory of all

user keys. Still, manual key verification, i. e., the link between public keys and the entities, such as hostnames or people, remains a challenging subject, especially in decentralized systems without pre-defined authorities, such as SSH, OpenPGP, and secure messaging [12, 41].

Many traditional authentication systems still rely on manual key-fingerprint comparisons [17]. Here, key-fingerprints are generated by encoding the (hashed) public key material into a human readable format, usually encoded in hexadecimal representation. A variety of alternatives such as QR Codes, visual fingerprints, Near Field Communication (NFC), and Short Authentication Strings (SAS) have been proposed. Most of these systems offer specific benefits, e. g., QR codes and NFC do not require users to compare strings, but they also come with specific disadvantages, e. g., they require hardware and software support on all devices. While advances are being made in these areas, the text-based representation is still the dominant form in most applications.

However, due to the recent boom of secure messaging tools, the debate of how to best represent and evaluate textual fingerprints has opened up again and there are many very active discussions among security experts [28, 33]. In April 2016, WhatsApp serving over one billion users enabled end-to-end encryption as default by implementing the Signal protocol. Key verification is optional and can be done by using QR codes or comparing numeric representations, in their case 60-digit numbers [43]. However, it is not clear whether their solution is more usable than traditional representations.

In this paper, we present an evaluation of different textual key-fingerprint representation schemes to aid in the secure messenger discussion. The requirements posed to the developers are as follows:

- The fingerprint representation scheme should provide offline support and work asynchronously. One reason for this is that fingerprints are often printed on business cards or exchanged by third parties.

- The fingerprint should be transferable via audio channels, e.g., it should be possible to compare fingerprint over the phone.
- The representation scheme should be as technically inclusive as possible. No special hardware or software should be required to verify the fingerprints; both require a concerted and coordinated effort between many actors to get enough coverage for a comparison mechanism to be worthwhile for users to adopt.
- The representation should be as inclusive as possible, i.e., excluding as few people with sensory impairments (visual, color, audio, etc.) as possible.

The above requirements exclude many proposed representation schemes and offer an explanation why they have not seen any adoption outside of academia. For this reason, we focus exclusively on textual fingerprint representations in our study. Textual key-fingerprints do not require hardware support and work in synchronous and asynchronous scenarios, i.e., they can be compared via voice or printed on business cards. Depending on the scheme, they even could be recalled from memory and exchanged over a voice channel.

This paper presents our study testing the usability of various textual key-fingerprint representation schemes. Our study consists of two parts: (1) an experiment where we measured how fast and accurate participants perform for different schemes, and (2) a survey about their perception and sentiment. These also contained a direct comparison between the representations.

Our findings suggest that the most adopted alphanumeric approaches such as the Hexadecimal and Base32 scheme perform worse than other alternatives: under a realistic threat model, more than 10% of the users failed to detect attacks targeting Hexadecimal representations, whereas our best system had failure rates of less than 3%. While the best system for accuracy is not the fastest, it is the system which received the highest usability rating and is preferred by users.

In the following sections, we discuss related work followed by an analysis of current implementations deploying in-persona key-fingerprint representation techniques and discuss our evaluated representation schemes. Then, we describe our experiment evaluating text-based key-fingerprint verification techniques with regards to their attack-detection accuracy and speed. Our experiment was conducted as an online study with 1047 participants recruited via the Amazon Mechanical Turk (MTurk) platform. We consider the scenario outlined above, where a user compares two key-fingerprint strings encoded by the different representation schemes. In addition to the implicit measurements of accuracy and speed, we also

```
alice@localhost:~$ ssh alice@example.com
The authenticity of host 'example.com (93.184.216.34)' 
can't be established.
RSA key fingerprint is
6f:85:66:da:e3:7a:02:c6:5e:62:3f:36:b7:d9:b4:2c.
Are you sure you want to continue connecting (yes/no)?
```

(a) OpenSSH: Lowercase Hexadecimal with Colons

```
alice@localhost:~$ gpg --fingerprint Bob
pub 2048R/00012282 2015-01-01 [expires: 2020-01-01]
      Key fingerprint =
      73EE 2314 F65F A92E C239  OD3A 718C 0701 0001 2282
uid                               Bob <bob@example.com>
```

(b) GnuPG: Uppercase Hexadecimal with Spaces

Figure 1: Alphanumeric Fingerprints Used in Practice

evaluate the self-reported user perception to get feedback about which systems are preferred by end users. Finally, we present our results, discuss their implications and takeaways, and conclude our work.

2 Related Work

Various key-fingerprint representations have been proposed in academia and industry. Various cryptographic protocol implementations still rely on manual fingerprint comparisons, while the hexadecimal representation is used in most of them. However, previous work suggests that fingerprint verifications are seldom done in practice [17, 37].

2.1 Key-Fingerprint Representations

Previous work has shown that users struggle with comparing long and seemingly “meaningless” fingerprints and it is suspected that they even might perform poorly in this task [19]. While most previous work has focused on the family of visual fingerprints [35, 32, 19, 10], to our knowledge, none of those focused on the differences between various different textual fingerprint representations.

Hsiao et al. have conducted a study with some textual and visual representation methods for hash verification [19]. They compared Base32 and simple word list representations with various algorithms for visual fingerprints and hash representation with Asian character sets (a subset of Chinese, Japanese Hiragana, and Korean Hangul, respectively). A within-subjects online study with 436 participants revealed that visual fingerprints score very well in both accuracy and speed, together with the Base32 text representation. Hsiao et al. conclude that depending on the available computation power and display size, either Base32 or one of the visual fingerprinting schemes should be used. They explicitly did not evaluate hexadecimal representation or digits

“because that scheme is similar to Base32 and known to be error-prone” [19]. However, our work shows that numeric representations actually perform significantly better than Base32 and is less error prone. In addition, our results suggest that language-based schemes, e.g., generated sentences achieve excellent results comparable to visual schemes. At the same time, textual approaches are more flexible (can be read out loud) and do not exclude people with sensory impairments.

Another study by Olembo et al. also focused mainly on the topic of visual fingerprints [32]. They developed a new family of visual fingerprints and compared them against a Base32 representation. The Base32 strings were twelve characters long and displayed without chunking. The participants performed better with the visual fingerprints than with Base32, regarding both accuracy and speed. Olembo et al. conclude that the Base32 representation is far away from optimal when it comes to manual key-fingerprint verification. We test this claim by comparing Base32 representation with other textual key-fingerprint representation and eventually prove it wrong.

Regarding chunking, Miller et al. have published *The magical number seven* and succeeding work that shows that most people can recall 7 ± 2 items from their memory span [29]. It has been shown that although there are slight differences between numbers, letters and words (numbers perform slightly better than letters, and letters slightly better than words), they perform similar in studies. More recent studies have shown that human working memory easily remembers up to 6 digits, 5.6 letters and 5.2 words [1, 6, 8]. Adjusting chunk sizes to these numbers can help users when comparing hashes.

While all of the above studies offer interesting insights into different (mainly visual) fingerprint representations, to the best of our knowledge there is not work focusing on which textual representation performs the best. However, this knowledge would be extremely important to help in the current debate in the secure messaging community. The representations currently being put forward and implemented are far from optimal and the results of our study can help improve the accuracy and usability of fingerprint representations. Unlike the above studies we conduct our study with a more realistic attacker strength, as presented in subsection 4.1).

2.2 Passwords and Passphrases

A passphrase is basically a password consisting of a series of words rather than characters. In academic literature, passphrases are often considered as a potentially more memorable and more secure alternative to passwords and are often recommended by system administrators [23, 40]. In contrast to most passphrase-

Scheme	Example
Hexadecimal	18e2 55fd b51b c808 601b ee5c 2d69
Base32	ddrf 17nv dpea qya3 5zoc 22i
Numeric	2016 507 6420 1070 394 1136 2973 991 70
PGP	locale voyager waffle disable Belfast performance slingshot Ohio spearhead coherence hamlet liberty reform hamburger
Peerio	bates talking duke rummy slurps iced farce pound day
Sentences	Your line works for this kind power cruelly. That lazy snow agrees upon our tall offer.

Table 1: Examples for different textual key-fingerprint representations for the same hash value

based systems, key-fingerprints cannot be chosen by the end-user and thus are more related to the system-assigned passphrases field: Bonneau et al. have shown that users are able to memorize 56-bit passwords [4]. miniLock¹ and its commercial successor Peerio² use system-assigned passphrases to generate cryptographic key pairs easing key backup and synchronization among multiple devices.

Contrary to widespread expectations, Shay et al. were not able to find any significant recall differences between system-assigned passphrases and system-assigned passwords [40]. However, they reported reduced usability due to longer submission times due to typing.

Similar to passphrases, the usage of language-based key-fingerprint representations is claimed to provide better memorability than just an arbitrary series of character strings despite the lack of empirical evidence. In our study, we measure the performance of the different approaches and also collect perception and feedback from end users.

3 Background

In the past years, various textual key-fingerprint representations have been proposed. In this section, we analyze currently practised in-persona key verification techniques in well-known applications. For comparison, Table 1 lists the approaches we used in our evaluation generated from the same hash value.

Only applications requiring manual key-fingerprint

¹<https://minilock.io>

²<https://peerio.com>

verification are considered. In mechanisms like S/MIME or X.509, fingerprints play only a secondary role because certificates are verified via certificate chains.

In the following, $SHA-1(x)$ ¹⁶ defines the execution of 16 rounds of nested $SHA-1$ on x , a truncation to the left-most 16 bits is defined by $x[0, \dots, 16]$, and pk is used as an abbreviation for the values of a public key (differs for RSA, DSA, or ECC).

3.1 Numeric

Numeric representation describes the notation of data using only numeric digits (0-9). The primary advantage of a such system is that Arabic numerals are universally understood, and in addition, numeric key-fingerprints show a similarity to phone numbers. The encoding is achieved by splitting a binary hash into chunks of equal length and expressing each chunk as a decimal number, e.g., by simply switching the representation base from 2 to 10.

The messaging and data exchange application SafeSlinger³ implements this as a fallback scheme for unsupported languages [14]. A 24 bit SAS in SafeSlinger (cf. Figure 2a) can be expressed by three decimal encoded 8-bit numbers.

In the messaging platform WhatsApp, a fingerprint is calculated by $SHA-256(pk)^{5200}[0, \dots, 240]$. This fingerprint is split up into six chunks, where each chunk is represented by a five digits long number modulo 100,000 [43]. Concatenating this fingerprint with the fingerprint of the communication partner results in the displayed representation, e.g.,

```
77658 87428 72099 51303
34908 23247 95615 27317
09725 59699 62543 54320
```

3.2 Alphanumeric

Alphanumeric approaches use numbers and letters to represent data. Depending on the representation type and its parameters, the letters can be presented either in lower-case or in upper-case. The string can be chunked into groups of characters, which are usually of equal length. Chunking does not alter the information contained, while changing lower-case letters to upper-case letters (and vice versa) may does, depending on the coding scheme. Commonly used representations are Hexadecimal, Base32, and Base64.

3.2.1 Hexadecimal

Hexadecimal digits use the letters A-F in addition to numerical digits and are a common representation for key-fingerprints and primarily used in SSH and OpenPGP.

³<https://www.cylab.cmu.edu/safeslinger>

Note that the case of the letters do not make any difference. Regarding chunking, both spaces (cf. Figure 1b) and colons (cf. Figure 1a) are commonly used as separation characters.

Key fingerprints in OpenPGP version 4 are defined in RFC 4880 [7] by

```
Hex(SHA-1(0x99 || len || 4 || creation_time || algo || pk))
```

where len is the length of the packet, $creation_time$ is the time the key has been created and $algo$ is unique identifier for the public-key algorithm. While the inclusion of $creation_time$ makes sure that even two keys with the same key material have different fingerprints, it allows an attacker to iterate through possible past times to generate similar fingerprints skipping the key generation step [5]. The actual representation of OpenPGP fingerprints is not defined in RFC 4880, but most implementations chose to encode them in hexadecimal form, e.g., GnuPG displays them uppercase in 16 bit blocks separated by whitespaces with an additional whitespace after 5 blocks (cf. Figure 1b), e.g.,

```
73EE 2314 F65F A92E C239 0D3A 718C 0701 0001 2282
```

Other implementations, such as OpenKeychain, deviate only slightly, for example by displaying them lowercase or with colored letters to ease comparison but still provide compatibility with GnuPG.

SSH fingerprint strings, as defined in RFC 4716 and RFC 4253 [15, 44], are calculated by

```
Hex(MD5(Base64(algo || pk)))
```

where $algo$ is a string indicating the algorithm, for example “ssh-rsa”. Fingerprints are displayed as “hexadecimal with lowercase letters and separated by colons” [15] (cf. Figure 1a), e.g.,

```
6f:85:66:da:e3:7a:02:c6:5e:62:3f:36:b7:d9:b4:2c
```

3.2.2 Base32

Base32 uses the Latin alphabet (A-Z) without the letters O and I (due to the confusion with numbers 0 and 10). There is no difference between lower-case letters and upper-case letters. In addition, a special padding character “=” is used, since the conversion algorithm processes blocks of 40 bit (5 Byte) in size. The source string is padded with zeroes to achieve a compatible length and sections containing only zeroes are represented by “=” [20, 21].

The ZRTP key exchange scheme for real-time applications is based on a Diffie-Hellman key exchange extended by a preceding hash commitment that allows for very short fingerprints, called Short Authentication

Strings (SAS) without compromising security [45]. The Base32 encoding used in ZRTP uses a special alphabet to produce strings that are easier to read out loud. VoIP applications such as CSipSimple⁴ use this Base32 option, usually named “B32” inside the protocol. Here, the leftmost 20 bits of the 32 bit SAS value are encoded as Base32. , e. g.,

5 e m g

3.2.3 Base64

There exist a number of specifications for encoding data into the Base64 format, which uses the Latin alphabet in both lower-case and upper-case (a-z, A-Z) as well as the digits 0-9 and the characters “+”, “/”, and “=” to represent text data. Again, the character “=” is used to encode padded input [20]. Starting with OpenSSH 6.8 a new fingerprint format has been introduced that uses SHA-256 instead of MD5 and Base64 instead of hexadecimal representation. In addition the utilized hash algorithm is prepended, e. g.,

SHA256:mVPwvezndPv/ARoIadVY98vAC0g+P/5633yTC4d/wXE

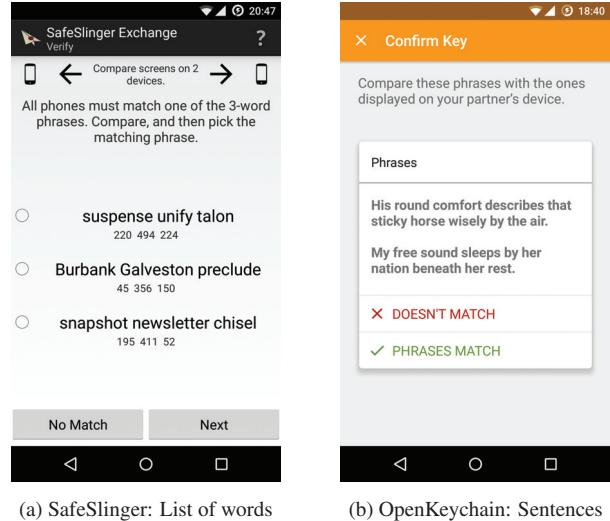
3.3 Unrelated Words

Instead of (alpha)numeric representation, fingerprints can be mapped to lists of words. Here, the binary representation is split into chunks, where each possible value of a chunk is assigned to a word in a dictionary. To increase readability, such a dictionary usually contains no pronouns, articles, prepositions and such. Word lists, such as the PGP Word List [22] and the Basic English word list compiled by K.C. Ogden [31], are primarily used for verification mechanisms based on SAS. Key-Fingerprints represented by words have been implemented for VoIP applications based on the ZRTP key exchange and other real-time communication protocols. Examples are Signal⁵, and the messaging and contact sharing application SafeSlinger [14] (cf. Figure 2). Besides their use in SAS based mechanisms, miniLock and Peerio utilize unrelated words for passphrase generation.

An example for a modern VoIP implementation that utilizes ZRTP for key exchange over Secure Real-Time Transport Protocol (SRTP) is Signal’s private calling feature, previously distributed as Redphone. The developers chose to implement only a specific subset of the ZRTP specification [45], namely Diffie-Hellmann key exchange via P-256 elliptic curves using “B256” SASs, i. e., Base256 encoding that maps to the leftmost 16 bits of the 32 bit SAS values to the previously introduced PGP Word List [22], e. g.,

⁴<https://github.com/r3gis3r/CSipSimple>

⁵<https://github.com/WhisperSystems/Signal-Android>



(a) SafeSlinger: List of words (b) OpenKeychain: Sentences

Figure 2: Language-based fingerprint representations

quota holiness

The messaging application SafeSlinger is based on a Group Diffie-Hellman protocol [14] implementing a key verification with SASs for up to 10 participants. In SafeSlinger the leftmost 24 bits of a SHA-1 hash is used to select 3 words from the PGP Word List, e. g.,

suspense unify talon.

Besides this, two other 3 word triples are selected to force users to make a selection before proceeding (cf. Figure 2a).

In contrast to Signal and SafeSlinger, Peerio (based on miniLock) does not use any SAS based verification mechanism. It uses pictures for verification and word lists for code generation. The word list is generated from most occurring words in movie subtitles. Besides key verification, these are also used to generate so called passphrases, which are used to derive their ECC private keys.

3.4 Generated Sentences

The words from the previous dictionaries can also be used to generate syntactically correct sentences as proposed by previous research: Goodrich et al. proposed to use a “syntactically-correct English-like sentence” representation for exchanging hash-derived fingerprints over audio by using *text-to-speech* (TTS) [16]. Michael Rogers et al. implemented a simple deterministic sentence generator [16, 38]⁶ Though the sentences from both approaches rarely make sense in a semantic fashion, they are syntactically correct and are claimed to pro-

⁶<https://github.com/akwizgran/basic-english>

vide good memorability. In our study, we used Michael Roger’s approach for our sentence generator.

We implemented this method for PGP fingerprints in OpenKeychain 3.6⁷ (cf. Figure 2b). To the best of the authors’ knowledge, to this date, it is the first integration of key verification via sentences although other projects are considering to change their fingerprint encoding scheme [38, 36].

4 Methodology

In order to evaluate the effect and perception of the different textual key-fingerprint representations, we conducted an online study on Amazon’s Mechanical Turk (MTurk) crowdsourcing service. Our Universities do not have an IRB, but the study conformed to the strict data protection law of Germany and informed consent was gathered from all participants. Our online study is divided into two parts: The experiment for performance evaluation followed by a survey extracting self-reported data from users. The survey ended with demographic questions.

4.1 Security Assumptions

In this section, we define the underlying security assumptions of our study, such as fingerprint method, length, and strength against an adversary. The fingerprint method and parameters are utilized consistently for all experiments in our study to offer comparability between all possible fingerprint representations. This attack model is important for the usability since an unrealistically strong or weak attacker could skew the results. Obviously, if the fingerprint strength is not kept equal between the systems this would also skew the results.

4.1.1 Fingerprint Method

To decide upon a fingerprint method for humanly verifiable fingerprints in our study, we first have to differentiate between human and machine verification to illustrate their differences. While a full fingerprint comparison can be implemented for machine verification, humans can fall for fingerprints that match only partially. Additionally, machine comparison can work with long values, whereas for human verification the length must be kept short enough to fit on business cards and to keep the time needed for comparison low.

For machine comparison, full SHA-256 hashes should be calculated binding a unique *ID* to the public key material. The probability of finding a preimage or collision attack is obviously negligible, but the fingerprints can still be computed fast in an ad-hoc manner when needed.

⁷<https://www.openkeychain.org>

It is important to note that collision resistance is not required for our scenarios. It is required for infrastructure-based trust models such as X.509, where certificates are verified by machines and trust is established by authority. In these schemes, a signature generated by a trusted authority can be requested for a certificate by proving the control over a domain, but then reused maliciously for a different certificate/domain. This is already possible with a collision attack, without targeting a full preimage. In contrast, the direct human-based trust schemes considered in this study only need to be protected against preimage attacks, because no inherently trusted authority is involved here.

While machine comparison needs to be done fast, e. g., on key import, manual fingerprint verification by humans is done asynchronously in person or via voice. Thus, we can use a key derivation function to provide a proof-of-work, effectively trading calculation time for a shorter fingerprint length. Secure messaging applications such as Signal or OpenPGP-based ones could pre-calculate the fingerprints after import and cache these before displaying them for verification later.

Thus, modern memory-hard key derivation functions such as *scrypt* [34] or *Argon2* [3] can be utilized to shorten the fingerprint length. These key derivation functions are parametrized to allow for different work factors. Suitable parameters need to be chosen by implementations based on their targeted devices and protocol.

As discussed in Section 3.2.1, while the generation of new fingerprints consists of the creation of a new key pair and the key derivation step, an attacker can potentially skip the key creation. Thus, in the following we only consider the key derivation performance as the limiting factor for brute force attacks.

When utilizing a properly parametrized key derivation function for bit stretching, the security of a 112 bit long fingerprint can be increased to require a brute force attack comparable to a classical 2^{128} brute force attacker. Consequently, a fingerprint length of 112 bit is assumed throughout our study.

4.1.2 Attacker Strength for Partial Preimages

In our user study, we assume an average attacker trying to impersonate an existing *ID* using our fingerprint method. Thus, an attacker would need to find a 112 bit preimage for this existing fingerprint using a brute force search executing the deployed key derivation function in each step. Due to the work factor, we consider this to be infeasible and instead concentrate on partial preimages. For comparability and to narrow the scope of our study, an attacker is assumed that can control up to 80 bits of the full 112 bit fingerprint.

Attackers might aim to find partial preimages where

the uncontrolled bits occur at positions that are more easily missed by inattentive users. First, the bits at the beginning and the end should be fixed as users often begin their comparison with these bits. Thus, we assume that, for any representation method, the first 24 and last 24 bits are controlled by the attacker and thus the same as in the existing fingerprint. Based on the feedback from our pre-study participants and reports from related work, this can be considered best-practice [17, 37]. Second, of the remaining 64 bits in the middle of our 112 bit fingerprint, we assume that 32 bits are controlled by the attacker in addition to the first 24 and last 24 bits. In total, we assume that 80 bits are controlled by the attacker, i. e., are the same as in the existing fingerprint, and 32 bit are uncontrolled.

The probability of finding such a partial preimage for a fingerprint when executing 2^{49} brute force steps is calculated approximately by

$$1 - \left(\frac{2^{112} - \sum_{k=1}^{32} \binom{64}{k}}{2^{112}} \right)^{2^{49}} \approx 0.66.$$

The inner parentheses of this equation define the probability that no partial preimage exists for one specific bit permutation. Instead of using $\binom{64}{32}$, a sum over 32 variations has been inserted to include permutations with more than the uncontrolled 32 bit that are also valid partial preimages. Finally, the probability to find a partial preimage is defined by the inverse of the exponentiation. Assuming the scrypt key derivation function parametrized with $(N, r, p) = (2^{20}, 8, 1)$, Percival calculates the computational costs of a brute force attack against 2^{38} ($\approx 26^8$) hashed passwords with \$610k and 2^{53} ($\approx 95^8$) with \$16B [34]. These costs can be considered a lower and upper bound for our attacker, which we assume to have average capabilities and resources. While 2^{38} has a probability of finding a partial preimage of only 0.05%, with 2^{42} the probability reaches nearly 1%, and with 2^{49} , as in our example, a partial preimage is found with over 50%.

In our study, we simulate attacks by inverting the bits from the existing fingerprint which are uncontrolled by the attacker, while the controlled bits are unchanged. For our theoretical approximation, we assume that the first 24 and last 24 bits should be controlled as well as 32 bits from the middle. In our study, we simulate an even more careful selection of appropriate fingerprints from the ones that an attacker would brute force. A general criteria here is to minimize the influence of uncontrolled bits on the entire fingerprint: For numeric and alphanumeric representations all bits affecting a character or digit are inverted together. For unrelated words, all bits affecting a word are changed. Sentences are never changed in a way that would alter the sentence structure.

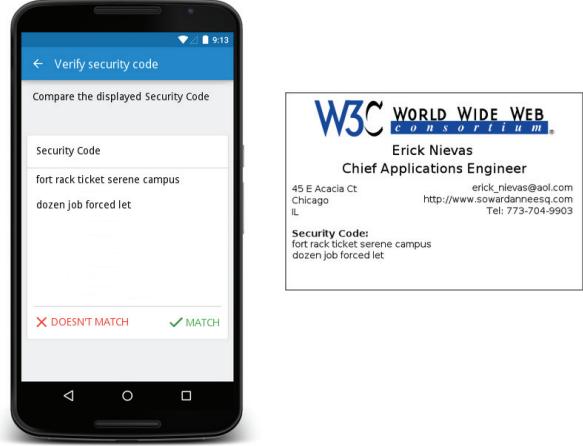


Figure 3: A screenshot of the actual task a user had to perform in the experiment. A user rates whether the *security codes* match, in this case with the Peerio word list approach, by clicking on the corresponding buttons shown on the phone.

4.2 Pre-Study

To get additional feedback from participants and evaluate our study design for flaws and misunderstandings, we conducted two small pre-studies: A lab study with 15 participants and an MTurk experiment with 200 participants, all required to perform 10 comparisons for each representation scheme (totally 60 comparisons in a randomized order). In our lab-study, we mainly focused on qualitative feedback, whereas the main goal of the MTurk pre-study was to find flaws in the presentation and task descriptions, as well as to check whether our proposed methodology is received as expected.

The biggest problem we found regarding the study design was that participants were uncertain if they should check for spelling mistakes in the words and sentence-based representation or if the all attacks would change entire words. To clarify this, a speech bubble was included in the task description that the participants do not have to look for spelling mistakes for language-based approaches.

We tested different rates of attack during the pre-study. The results showed that participants who were exposed to frequently occurring attacks were more aware and had a much higher attack detection rate. For our main study, we reduced the number of attacks to 40 comparisons with 4 attacks to have a good balance between true positives and false negatives. We received feedback that attacks on anchor parts of the strings, i. e., in the beginning, end, and at line breaks could be easily detected. Many users had problems with distinguishing the hexadecimal from the Base32 representation as well as distinguishing different word list approaches (Peerio vs. OpenPGP word list). Thus, we opted for a mixed factorial study

design where users test *only one* scheme of each type. We grouped the hexadecimal and Base32 scheme for the alphanumeric type and the PGP and Peerio for the word-list type together. These two groups were tested between-subjects in a split-plot design, i. e., the participants test either hexadecimal or Base32 for the alphanumeric type. See Table 2 for a graphical representation of our condition assignment design.

4.3 Experiment Design

The main part of our online study is the experiment part where users perform actual fingerprint comparisons. Here, we conducted two separate experiments with a distinct set of participants: (1) our main experiment testing different textual high-level representation schemes against each other and (2) a secondary experiment testing different chunk sizes for the hexadecimal representation. We opted for two distinct experiments due to the exponential growth of experiment conditions, as described in Section 4.3.1.

Before letting the participants start our experiment, we explained the scenario:

“With this HIT, we are conducting an academic usability study followed by a short survey about different types of security codes used in the IT world. Security codes are often used in encrypted communications to identify the participants in a communication. If the security codes match, you are communicating securely. If they don’t match, an eavesdropper may be intercepting your communication”.

On MTurk, the term *Human Intelligence Task*, or *HIT* stands for a self-contained task that a worker can work on, submit answers, and get a reward for completing. Since our participants might not be familiar with the *key-fingerprint representation* term, we replaced it with *security codes* for the sake of the study.

We opted not to obfuscate the goal of the study since our research aims at finding the best possible representation for the comparison of key-fingerprints in a security context. This is closest to how users interact with fingerprints in the real world — their secure messaging applications also ask them to compare the strings for security purposes. The question how to motivate users to compare fingerprints is an entirely different research question. So in our case, we believe it was not necessary or desirable to use deception and since deception should be used as sparingly as possible we opted for the “honest” approach.

After agreeing the terms, participants are shown a fictitious business card next to a mobile phone, both displaying a security code (as shown in Figure 3). To become more familiar with the task, the experiment is

Type (Within-Group)	Scheme (Between-Group)
Alphanumeric	Hexadecimal XOR Base32
Numeric	Numeric
Unrelated Words	PGP XOR Peerio
Generated Sentences	Generated Sentences

Table 2: To avoid confusion between too similar approaches (cf. Section 4.2), in our condition assignment, scheme types (left column) can consist of multiple representation schemes (right column). Each participant tests *only one* randomly assigned scheme of each type in a randomized order. .

started with 4 training tasks (each method once) not considered in the evaluation. The user’s only task is to rate whether the shown fingerprints match by clicking on *Match* or *Doesn’t Match* on the phone. Based on the condition assignment, participants see different approaches in a *randomized order*. We measure whether their answer was correct and their speed, i. e., the amount of time spent on the comparison. The experiment is concluded with a survey collecting feedback on the used approaches and the tasks and demographic information discussed in the “Results” section.

4.3.1 Variables and Conditions

In the main experiment, the used *representation scheme* is our controlled independent variable whereas its values define our experiment conditions. In our additional chunking experiment, the *chunking size* is our controlled independent variable instead of the representation algorithm. During all tasks, we measure how fast participants perform with their given conditions and whether they are able to detect attacks by rating “incorrect” (*speed* and *accuracy* as our measured dependent variables).

In both experiments, each user had to perform 46 comparisons in total. To detect users clicking randomly, 2 obviously distinct comparisons were added to test a participant’s attention. Training comparisons and attention tests are not included in the evaluation. Based on the feedback in our pre-study, we added tooltips during the training comparisons giving hints for language-based approaches telling the user that spelling attacks would not occur. We set the number of attacks to six: two obvious attacks where all bits are altered serving as control questions and 4 actual attacks with partial 80-bit preimages (one for each representation scheme). Participants failing at the control attacks are not considered in the evaluation but still received a payment if finishing all tasks. The major challenge in the study design is a high attack detection rate in general: most users perform comparisons correctly for the given attacker strength.

To avoid side effects, we chose fixed font size, color



Figure 4: A screenshot showing a statement rating in the post-experiment survey. Since the participants might not distinguish the different types, we have provided an example from their previous task.

and style, i. e., the same typeface for all fingerprint representations. In addition, we set fixed line breaks for sentences and word lists. In the main experiment, the same chunking style was used for all representations: For (alpha)numeric approaches a chunk consists of four characters separated by spaces. For word lists, we opted for a line break every four words. In the generated sentences representation, one sentence per line is displayed. We are aware that all these design decisions can have an effect on the comparison of the representations. However, our pre-study results show a significantly lower effect size. More importantly, we are mainly interested in comparing the concepts, therefore we did not vary any of the visual attributes like font size or style. In particular, differences resulting from the font's typeface have not been evaluated. Lund showed in his meta-analysis that there are no significant legibility differences between serif and sans serif typefaces [25].

Chunk-Size Testing A question was raised whether the chunking of a hexadecimal string plays a greater role in comparison to the different approaches. Thus, in addition to the main experiment testing different representation types, we conducted a second experiment with new participants testing different chunk sizes for the hexadecimal representation. Here, we used chunk-sizes ranging from 2 to 8 in addition to “zero-chunk size” (8 cases). The zero-chunk size means that no spaces have been included. To make the results more comparable, we opted for a similar design as done in the major experiment, i. e., we required the same amount of comparisons, used the same font settings, and had the same amount of attacks. For each participant, we assigned 4 out of 8 different chunk-sized randomly. Same as in the major experiment, all participants had to compare 46 fingerprints whereas the first 4 are considered as training comparisons, 4 attacks (one for each chunk size), and 2 control attacks with obviously distinct fingerprints.

The major experiment is followed by a survey fo-

cusing on self-reported user perception and opinions about the different approaches. This is the main reason we opted to compare as much as possible in a within-groups fashion and only selected a small number of conditions in total. Since users might not notice the difference between the various dictionary or alphabet approaches, we designed a mixed factorial design where the users would only get one of the alphabets/dictionaries (between-subjects) but they would test all different high-level systems (within-group) as depicted in Table 2. The between-group conditions have been assigned randomly with a uniform distribution. Since participants from our pre-study had difficulties to distinguish the different chunking approaches, we skipped the survey part in the chunk-size experiment.

4.3.2 Online Survey

The experiment was followed by an online survey gathering self-reported data and demographics from participants. To measure perception, we asked the participants whether they agreed with statements discussed in subsection 5.2 on a 5 point Likert scale: from strongly disagree to neural strongly agree as shown in Figure 4. Participants had to rate each representation type for all statements. Since users might not distinguish the different representation schemes, we provide an example from their previously finished task.

4.3.3 Statistical Testing

We opted for the common significance level of $\alpha = 0.05$. To counteract the multiple comparisons problem, we use the Holm-Bonferroni correction for our statistical significance tests [18]. Consequently, all our p-values are reported in the corrected version.

We test the comparison duration with the Mann-Whitney-Wilcoxon (MWW) test (two-tailed). We opt for this significance test due to a few outliers, consequently a

Scheme	Speed				Accuracy			Total		
	mean [s]	med [s]	stdev	p-val	fail-rate	p-val	f-pos	fails	attacks	tests
Hexadecimal	11.2	10.0	6.4		10.44		0.49	50	479	4765
<i>Hexadecimal – Base32</i>	1.0	1.1	0.0	<0.001	-1.94	0.690	-2.09	12	32	269
<i>Hexadecimal – Numeric</i>	0.6	0.5	0.6	<0.001	-4.10	0.048	0.21	-9	-452	-4527
<i>Hexadecimal – PGP</i>	-1.8	-1.2	-1.0	<0.001	-1.65	0.690	-0.01	11	35	340
<i>Hexadecimal – Peerio</i>	2.5	2.7	0.8	<0.001	-4.69	0.048	0.08	22	-8	-91
<i>Hexadecimal – Sentences</i>	-1.1	-0.7	-0.6	<0.001	-7.45	<0.001	-0.99	22	-457	-4518
Base32	10.2	8.9	6.4		8.50		2.58	38	447	4496
<i>Base32 – Hexadecimal</i>	-1.0	-1.1	-0.0	<0.001	1.94	0.690	2.09	-12	-32	-269
<i>Base32 – Numeric</i>	-0.4	-0.6	0.6	<0.001	-2.16	0.404	2.30	-21	-484	-4796
<i>Base32 – PGP</i>	-2.8	-2.3	-1.0	<0.001	0.28	0.714	2.08	-1	3	71
<i>Base32 – Peerio</i>	1.5	1.6	0.8	<0.001	-2.75	0.404	2.17	10	-40	-360
<i>Base32 – Sentences</i>	-2.1	-1.8	-0.6	<0.001	-5.51	<0.001	1.10	10	-489	-4787
Numeric	10.6	9.5	5.8		6.34		0.28	59	931	9292
<i>Numeric – Hexadecimal</i>	-0.6	-0.5	-0.6	<0.001	4.10	0.048	-0.21	9	452	4527
<i>Numeric – Base32</i>	0.4	0.6	-0.6	<0.001	2.16	0.404	-2.30	21	484	4796
<i>Numeric – PGP</i>	-2.4	-1.7	-1.6	<0.001	2.45	0.404	-0.22	20	487	4867
<i>Numeric – Peerio</i>	1.9	2.2	0.2	<0.001	-0.59	0.714	-0.13	31	444	4436
<i>Numeric – Sentences</i>	-1.7	-1.2	-1.2	<0.001	-3.35	0.004	-1.20	31	-5	9
PGP	13.0	11.2	7.4		8.78		0.50	39	444	4425
<i>PGP – Hexadecimal</i>	1.8	1.2	1.0	<0.001	1.65	0.690	0.01	-11	-35	-340
<i>PGP – Base32</i>	2.8	2.3	1.0	<0.001	-0.28	0.714	-2.08	1	-3	-71
<i>PGP – Numeric</i>	2.4	1.7	1.6	<0.001	-2.45	0.404	0.22	-20	-487	-4867
<i>PGP – Peerio</i>	4.3	3.9	1.8	<0.001	-3.03	0.337	0.09	11	-43	-431
<i>PGP – Sentences</i>	0.7	0.5	0.4	<0.001	-5.79	<0.001	-0.98	11	-492	-4858
Peerio	8.7	7.3	5.6		5.75		0.41	28	487	4856
<i>Peerio – Hexadecimal</i>	-2.5	-2.7	-0.8	<0.001	4.69	0.048	-0.08	-22	8	91
<i>Peerio – Base32</i>	-1.5	-1.6	-0.8	<0.001	2.75	0.404	-2.17	-10	40	360
<i>Peerio – Numeric</i>	-1.9	-2.2	-0.2	<0.001	0.59	0.714	0.13	-31	-444	-4436
<i>Peerio – PGP</i>	-4.3	-3.9	-1.8	<0.001	3.03	0.337	-0.09	-11	43	431
<i>Peerio – Sentences</i>	-3.6	-3.4	-1.4	<0.001	-2.76	0.075	-1.07	0	-449	-4427
Sentences	12.3	10.7	7.0		2.99		1.48	28	936	9283
<i>Sentences – Hexadecimal</i>	1.1	0.7	0.6	<0.001	7.45	<0.001	0.99	-22	457	4518
<i>Sentences – Base32</i>	2.1	1.8	0.6	<0.001	5.51	<0.001	-1.10	-10	489	4787
<i>Sentences – Numeric</i>	1.7	1.2	1.2	<0.001	3.35	0.004	1.20	-31	5	-9
<i>Sentences – PGP</i>	-0.7	-0.5	-0.4	<0.001	5.79	<0.001	0.98	-11	492	4858
<i>Sentences – Peerio</i>	3.6	3.4	1.4	<0.001	2.76	0.075	1.07	0	449	4427

Table 3: Our experiment results showing the differences between the representation schemes. The top rows of each row group separated by a rule, show the raw performance of a baseline scheme, followed by italic rows showing a direct comparison delta. Greyed-out values are not backed by statistical significance. The columns *fail-rate* (undetected attacks) and *false-pos* (same string rated as an attack) display percentage values.

slightly skewed normal distribution, and a large amount of collected data. The *common language effect size* is shown by mean and median comparisons [26].

The attack detection rate is tested with a pairwise Holm-Bonferroni-corrected Barnard’s exact test (Exakt package in R) achieving one of highest statistical power for 2x2 contingency tables [2].

Survey ratings are, again, tested by using the MWU significance test (two-tailed test). As has been shown in previous research [9], it is most suitable for 5-point Likert scales, especially if not multimodal distributed as in our survey results. In case two fingerprint representation schemes are statistically tested against each other, only participants encountering both schemes were considered.

5 Results

In this section, we present our results: our online study with 1047 participants has been conducted in August and September 2015. The study for testing the chunk size has been conducted in February 2016 with 400 participants. Starting with our online experiment evaluation showing the raw performance of users, we then present user perception results from the follow-up survey. Finally, we discuss the demographics of our participants.

5.1 Online Experiment

Participants who have not finished all comparisons or failed the attention tests were excluded from our eval-

Scheme	Speed			Accuracy			Total		
	mean [s]	med [s]	p-val	fail-rate	p-val	false-pos	fails	attacks	tests
Hexadecimal (4)	12.3	10.4		6.78		0.38	16	236	2360
<i>hex (4) – hex (0)</i>	-2.4	-2.6	<0.001	0.33	1.000	-0.28	-2	-17	-170
<i>hex (4) – hex (2)</i>	-0.3	-0.9	<0.001	1.37	1.000	0.00	-3	3	30
<i>hex (4) – hex (3)</i>	-0.3	0.1	0.362	-0.64	1.000	0.09	2	8	80
<i>hex (4) – hex (5)</i>	-1.4	-1.2	<0.001	1.01	1.000	-0.40	-2	5	50
<i>hex (4) – hex (6)</i>	-1.9	-1.8	<0.001	2.43	1.000	0.09	-5	8	80
<i>hex (4) – hex (7)</i>	-1.7	-1.8	<0.001	3.35	1.000	0.19	-8	-1	-10
<i>hex (4) – hex (8)</i>	-2.8	-3.2	<0.001	1.35	1.000	-0.12	-4	-10	-100

Table 4: Comparison of the chunking experiment results showing the differences between the representation schemes. The top row shows the raw performance of the hexadecimal scheme with a four-character chunking, followed by italic rows showing a direct comparison delta. Greyed-out values are not backed by statistical significance. The columns *fail-rate* (undetected attacks) and *false-pos* (same string rated as an attack) display percentage values.

uation: all participant compared 46 security codes in a randomized order, whereas 40 (10 of each scheme) were considered in the evaluation. The four training samples and the control questions are excluded. Few comparisons done in less than 2 seconds and more than one minute have been excluded. The reason for such can either be multiple clicks during the page load, or external interruptions of the participants. None of the attack could be successfully detected in under 4 seconds.

Our experiment results, summarized in Table 3, show the raw performance of all schemes regarding their speed, accuracy and false-positive rate. The top rows of each row group, separated by a rule, show the raw performance of a representation scheme as baseline (negative values indicate lower values than the baseline). The following rows show a direct comparison delta between between two schemes. The speed column group consists of the mean and median (in seconds), the standard deviation and the according p-values for a direct comparison. The fail-rate column shows the rate of the undetected attacks with the according p-values for a direct comparison. The total column group simply shows the total numbers of tests, attacks and undetected attacks.

The results show that the average time spent on comparisons plays only a minor role among the schemes: 4.3 s difference between the best and the worst scheme. Note that the Peerio word-list scheme performed best with 8.7 s mean whereas the PGP word list performed worst with 13 s mean ($p < 0.001$).

However, there is a clear effect regarding the attack detection rate (see Table 3). All alternative key-fingerprint representations performed better than the state-of-the-art hexadecimal representation, where 10.1% of attacks have not been detected by the users. Previous work shows similar numbers for Base32 [19]. To our surprise, the numeric approach performs better in both categories: it features an attack detection rate of 93.57% ($p < 0.01$) and an average speed of 10.6s ($p < 0.001$). Generated sentences achieved the highest attack detection rate of

97.97% with a similar average speed as the hexadecimal scheme. On the downside, this scheme has produced a slightly higher false-positive rate. We found that the false positives occurred mostly with longer sentences where there has been a line break on the phone mock-up due to portrait orientation. This is a realistic problem of this system if used with portrait orientation and not a problem with our mock-up in itself. Improvements on making the sentences shorter could mitigate this situation.

Chunk-Size Experiment

Table 4 summarizes the results of our secondary chunk-size experiment. As can be seen, no statistically significant results have been achieved for the *attack detection fail-rate* (undetected attacks by end users). However, we observed that the chunk sizes with 3 and 4 characters performed best in speed, even though the effect sizes were minor: only 3.3 seconds difference with similar standard deviations between the best and worst chunk size setting.

Firstly, we notice that despite the same attack strength as in our major experiment, participants were able to detect more attacks. We suspect that the higher attack detection rate is based on (1) a higher learning effect due to the same scheme for all comparisons and (2) in contrast to our major study, participants had a slightly higher drop-out rate and thus only more motivated participants were considered. This is supported by the numbers in the total tests column of Table 4: here, we can see that for the zero-chunking and chunking with 8 characters less tests have been performed. This is based on the fact that although the chunk sizes have been assigned almost uniformly, participants assigned with harder chunk settings often dropped out before even finishing their entire task.

More importantly, our results also support the claim from our pre-study: The chunking parameter in hexadecimal strings plays only a minor role in the *attack detection fail-rate*.

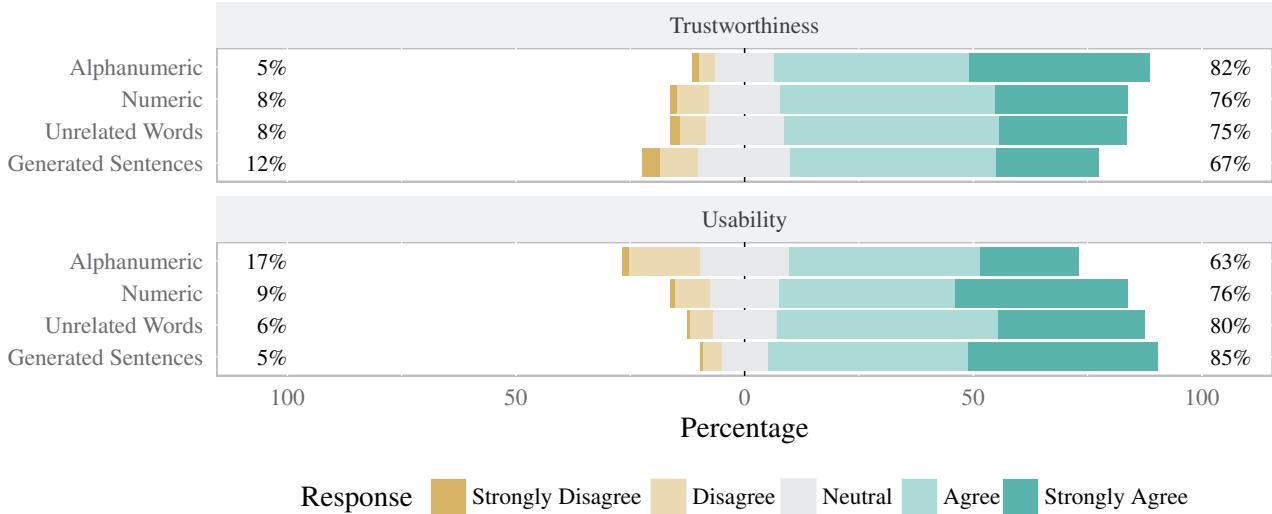


Figure 5: Aggregated survey results for statement rating regarding the usability and trustworthiness.

5.2 Online Survey

To measure the usability and trustworthiness of all representation schemes, we asked our participants whether they agreed with the following statements:

- S_1 The comparisons were easy for me with this method
- S_2 I am confident that I can make comparisons using this method without making mistakes
- S_3 I think making comparisons using this method would help me keep my communications secure
- S_4 I was able to do the comparisons very quickly with this method
- S_5 I found this method difficult to use
- S_6 Overall, I liked this method

We mixed positive and negative statements, e.g., S_1 and S_5 , to create a more robust measure. S_6 is used to calculate the overall ranking of the different representation schemes.

Figure 5 shows the aggregated results where the usability statements are grouped to one usability feature and the trustworthiness derived from the rating on the statement S_3 . Negative statement ratings have been inverted for a better comparison. Figure 6 shows the rating results for each specific statement in the survey. The order of the tested schemes has been chosen randomly, but was kept consistent across all statements. Same as in our online experiment evaluation, the pairwise statistical

tests are Holm-Bonferroni corrected. In case of a direct statistical test between two schemes, only users encountering both schemes have been considered. All in all, the usability perception of the participants is almost consistent with the performance results from the experiment.

To measure the perception of the task difficulty, we asked the participants whether they agreed with the statements S_1 , S_2 and S_3 respectively. As illustrated in Figure 6 in the Appendix A, the effect size between the different approaches is low. However, the participants were more likely to agree that language-based representation schemes are easier to use. For instance, we see that in comparison to the alphanumeric schemes (average rating of 3.4), word list (average rating of 3.9, $p < 0.001$) and generated sentence schemes (average rating of 4.2, $p < 0.001$) are rated to be easier by our participants (S_1 , S_5). While the experiment results of the sentence generators clearly outperformed all other approaches, they also were rated better by the participants. Same applies for the low-performing hexadecimal and Base32 schemes which clearly received lower ratings. Consistently with the surprising performance results in the experiment, the numeric scheme is also considered to be easier by many participants: average rating of 3.9 and $p < 0.001$.

The sentence generator scheme achieved the highest user confidence rating “making comparisons without any mistakes” (S_2 , $p < 0.001$ for all pairwise comparisons). The participants’ perception is consistent with the experiment results where the word-list-based and sentence generator schemes lead to higher attack detection rates.

The ratings for S_4 illustrate that more complex representation schemes from the user’s point of view, such as hexadecimal and Base32, are considered to be more secure by participants, even though all approaches provide the same level of security.

5.3 Demographics

A total of 1047 users participated in the online study while only 1001 have been considered in the evaluation due to our two control questions. Out of the evaluated participants, 534 participants were male, 453 were female, 4 chose other while the rest opted to not give any information. No significant difference between genders could be found, with a subtle trend of a higher accuracy for women and higher speed among men. The median age was 34 (34.4 average) years, while 34 participants chose not to answer (no statistically significant differences between ages).

A total of 39 people reported to have “medical conditions that complicated the security code comparisons (e.g., reading disorders, ADHD, visual impairments, etc.)” with a slightly higher undetected attack rate (statistically insignificant due to small sample size and thus low statistical power).

The majority of the participants stated to have a Bachelor’s degree (399 of 1047) as their highest education whereas 34% chose not to answer. 931 participants have started our HIT but stopped early during the experiment (mostly after the first few comparisons). 160 users reported the general task to be annoying.

6 Discussion

The results of our study show that while there are subtle speed variations among all approaches, the attack detection rate and user perception for the current state-of-the-art hexadecimal key-fingerprint representation is significantly lower than those of most alternative representation schemes. Language-based representations (with the exception of the PGP word list) show improved user behaviour leading to a higher detection rate of attacks. To improve the usability of key-fingerprints, we propose the following takeaways based on our study results.

6.1 Takeaways

Our results show that all representation schemes achieve a high accuracy (high attack detection rate) and can be performed quickly by users. As expected, language-based fingerprint representations are more resilient against attacks (higher attack detection rate) and achieve better usability scores. Among all conditions, alphanumeric approaches performed worse and have been out-

performed. For instance, the *numeric* representation was more suitable than hexadecimal and Base32. The raw performance results suggest a similar speed for the numeric representation with a higher attack detection rate, and it also has received better usability ratings from end-users.

Our chunking experiment has shown that chunk-sizes play only a minor role in improving attack detection rates (we could not find statistically significant differences). However, if a hexadecimal representation is used chunks of 3 and 4 characters perform best.

As shown by the word list representations, the comparison speed can be increased by larger dictionaries leaving room for improvement in this area. Even though all representation schemes provide the same level of security, exotic looking solutions are considered to be more secure by end users.

6.2 Limitations

Most importantly, our study design *does not test* whether end users *are actually willing to compare any fingerprints* in practice. We only aim to study how easy different representations are to compare from the users’ point of view.

As with any user study conducted with MTurk, there is concern about the external validity of the results: users in the real world might show different behaviour. This is mainly because of two reasons: (1) in practice fingerprint comparisons will seldom occur in a such frequency, and (2) when performed in practice play a more important role than just participating in an anonymous online study. Additionally, MTurkers have been shown to be more tech-savvy and are better in solving textual and visual tasks in comparison to the average population. Thus, they could have performed better in most of the comparison conditions than the average population. It is also known that some MTurkers just “click through” studies to get the fee and thus distort study results. Our counterbalanced study design with included control questions and statistical significance tests mitigate this effect. For instance, we excluded 46 out of 1047 participants from our main study part based on these questions being answered incorrectly.

Due to the within-group part of our factorial design, many parameter choices such as different fonts, font sizes, attack rates, etc. could not be considered. These are, however, interesting avenues for future work. As shown in our additional chunking experiment, another challenge in testing different parameters is the high attack detection rate, where subtle changes would require a high amount of users to produce statistically significant results.

Due to the anonymous nature of online studies, it is

also impossible to reliably tell which languages a participant is fluent in. We specified that we only wanted participants from English-speaking countries, however we had no way of checking compliance except by relying on self-reported data. Language-based representation approaches might induce additional barriers for non-native speakers, e. g., due to unknown or unfamiliar words.

7 Conclusion and Future Work

We evaluated six different key-fingerprint representation types with regards to their comparison speed, attack detection accuracy and usability, which encompasses attack detection but also resilience against human errors in short-term memory. An online study with 1047 participants was conducted to compare numeric, alphanumeric (Hexadecimal and Base32), word lists (PGP and Peerio), as well as generated sentences representation schemes for key-fingerprint verification. All fingerprint representations were configured to offer the same level of security with the same attacker strength.

Our results show that usage of the large word lists (as used in Peerio) lead to the fastest comparison performance, while generated sentences achieved highest attack detection rates. In addition, we found that additional parameters such as chunking of characters plays only a minor role in the overall performance. The widely-used hexadecimal representation scheme performed worst in all tested categories which indicates that it should be replaced by more usable schemes. Unlike proposals which call for radically new fingerprint representations, we studied only textual fingerprint representations, which means that the results of our work can be directly applied to various encryption applications with minimal changes needed. Specifically, no new hardware or complex software is required: applications merely need to replace the strings they output to achieve a significant improvement in both attack-detection accuracy and usability.

There are various interesting areas of future work. Firstly, we chose to study only a selected sample from the design space of fingerprint representations in a within-subjects design, so we could facilitate a direct comparison between the different classes of fingerprints. Further work exploring line breaks, font settings, dictionaries, different attacker strengths, etc. will likely lead to further improvement possibilities.

While this work shows that there are better ways to represent key-fingerprints than currently being used, it does not explore what can be done to motivate more users to actually compare the fingerprints in the first place. Follow-up studies to research this important question are naturally an interesting and vital area of research.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments, and Trevor Perrin, Jake McGinty, Tom Ritter and Skylar Nagao for their discussion and excellent feedback.

References

- [1] BADDELEY, A. Working memory. *Science* 255, 5044 (1992), 556–559.
- [2] BARNARD, G. Significance tests for 2×2 tables. *Biometrika* (1947), 123–138.
- [3] BIRYUKOV, A., DINU, D., AND KHOVRATOVICH, D. Argon2: the memory-hard function for password hashing and other applications. Tech. rep., Password Hashing Competition (PHC), December 2015.
- [4] BONNEAU, J., AND SCHECHTER, S. Towards reliable storage of 56-bit secrets in human memory. In *Proceedings of the 23rd USENIX Security Symposium* (August 2014).
- [5] BREITMOSER, V. pgp-vanity-keygen. <https://github.com/Valodim/pgp-vanity-keygen>, 2014.
- [6] BUCKNER, R. L., PETERSEN, S. E., OJEMANN, J. G., MIEZIN, F. M., SQUIRE, L., AND RAICHLE, M. Functional anatomical studies of explicit and implicit memory retrieval tasks. *The Journal of Neuroscience* 15, 1 (1995), 12–29.
- [7] CALLAS, J., DONNERHACKE, L., FINNEY, H., SHAW, D., AND THAYER, R. OpenPGP Message Format. RFC 4880 (Proposed Standard), Nov. 2007. Updated by RFC 5581.
- [8] CRANNELL, C., AND PARRISH, J. A comparison of immediate memory span for digits, letters, and words. *The Journal of Psychology* 44, 2 (1957), 319–327.
- [9] DE WINTER, J. C., AND DODOU, D. Five-point Likert items: t test versus Mann-Whitney-Wilcoxon. *Practical Assessment, Research & Evaluation* 15, 11 (2010), 1–12.
- [10] DHAMIJA, R. Hash visualization in user authentication. In *CHI '00 Extended Abstracts on Human Factors in Computing Systems* (New York, NY, USA, 2000), CHI EA '00, ACM, pp. 279–280.
- [11] DIERKS, T., AND RESORLA, E. The Transport Layer Security Protocol Version 1.2. RFC 5246, Aug. 2008. Updated by RFCs 5746, 5878, 6176.
- [12] ELECTRONIC FRONTIER FOUNDATION. Secure Messaging Scorecard. <https://www.eff.org/secure-messaging-scorecard>, 2014.
- [13] ELLISON, C., ET AL. Establishing identity without certification authorities. In *USENIX Security Symposium* (1996), pp. 67–76.
- [14] FARB, M., LIN, Y.-H., KIM, T. H.-J., MCCUNE, J., AND PERRIG, A. Safeslinger: easy-to-use and secure public-key exchange. In *Proceedings of the 19th annual international conference on Mobile computing & networking* (2013), ACM, pp. 417–428.
- [15] GALBRAITH, J., AND THAYER, R. The Secure Shell (SSH) Public Key File Format. RFC 4716 (Informational), Nov. 2006.
- [16] GOODRICH, M. T., SIRIVANOS, M., SOLIS, J., TSUDIK, G., AND UZUN, E. Loud and clear: Human-verifiable authentication based on audio. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on* (2006), IEEE, pp. 10–10.
- [17] GUTMANN, P. Do users verify SSH keys? *USENIX;login:* 36, 4 (2011).

- [18] HOLM, S. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.
- [19] HSIAO, H.-C., LIN, Y.-H., STUDER, A., STUDER, C., WANG, K.-H., KIKUCHI, H., PERRIG, A., SUN, H.-M., AND YANG, B.-Y. A study of user-friendly hash comparison schemes. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual* (2009), IEEE, pp. 105–114.
- [20] JOSEFSSON, S. The Base16, Base32, and Base64 Data Encodings. RFC 3548 (Informational), July 2003.
- [21] JOSEFSSON, S. The Base16, Base32, and Base64 Data Encodings. RFC 4648, Oct. 2006.
- [22] JUOLA, P. Whole-word phonetic distances and the PGPFone alphabet. In *Spoken Language, 1996. ICSLP 96. Proceedings., Fourth International Conference on* (1996), vol. 1, IEEE, pp. 98–101.
- [23] KEITH, M., SHAO, B., AND STEINBART, P. A behavioral analysis of passphrase design and effectiveness. *Journal of the Association for Information Systems* 10, 2 (2009), 2.
- [24] LAURIE, B., LANGLEY, A., AND KASPER, E. Certificate Transparency. RFC 6962 (Experimental), June 2013.
- [25] LUND, O. *Knowledge construction in typography: the case of legibility research and the legibility of sans serif typefaces*. PhD thesis, The University of Reading, Department of Typography & Graphic Communication., 1999.
- [26] McGRAW, K. O., AND WONG, S. A common language effect size statistic. *Psychological bulletin* 111, 2 (1992), 361.
- [27] MELARA, M. S., BLANKSTEIN, A., BONNEAU, J., FREEDMAN, M. J., AND FELTEN, E. W. CONIKS: A Privacy-Preserving Consistent Key Service for Secure End-to-End Communication. Tech. Rep. 2014/1004, Cryptology ePrint Archive, December 2014.
- [28] [MESSAGING] MAILING-LIST ARCHIVE. Usability of Public-Key Fingerprints. <https://moderncrypto.org/mail-archive/messaging/2014/00004.html>, 2014.
- [29] MILLER, G. A. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review* 63, 2 (1956), 81.
- [30] NAMECOIN PROJECT. Namecoin. <http://namecoin.info>, Nov. 2014.
- [31] OGDEN, C. K., ET AL. System of Basic English. *Self-published* (1934).
- [32] OLEMBO, M. M., KILIAN, T., STOCKHARDT, S., HÜLSING, A., AND VOLKAMER, M. Developing and testing a visual hash scheme. In *EISMIC* (2013), pp. 91–100.
- [33] [OPENPGP] IETF MAIL ARCHIVE. Fingerprints. <https://mailarchive.ietf.org/arch/msg/openpgp/2C9gTsxTgh29W8VX8x700YZqfUY>, 2015.
- [34] PERCIVAL, C. Stronger key derivation via sequential memory-hard functions. *Self-published* (2009).
- [35] PERRIG, A., AND SONG, D. Hash visualization: A new technique to improve real-world security. In *International Workshop on Cryptographic Techniques and E-Commerce* (1999), pp. 131–138.
- [36] PINGEL, I., IRVING, A., GENERALMANAGER, WIKINAUT, TIN-LOAF, FARB, M., AND JPOPPLEWELL. Fingerprint exchange - issue #826 - whispersystems/textsecure - github.
- [37] PLASMOID. Fuzzy Fingerprints: Attacking Vulnerabilities in the Human Brain. <http://www.thc.org/papers/ffp.html>, Oct. 2003.
- [38] ROGERS, M., AND PERRIN, T. Key-Fingerprint Poems. <https://moderncrypto.org/mail-archive/messaging/2014/000125.html>, 2014.
- [39] RYAN, M. D. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS* (2014), NDSS.
- [40] SHAY, R., KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., UR, B., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Correct horse battery staple: Exploring the usability of system-assigned passphrases. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (2012), ACM, p. 7.
- [41] UNGER, N., DECHAND, S., BONNEAU, J., FAHL, S., PERL, H., GOLDBERG, I., AND SMITH, M. Sok: Secure messaging. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 232–249.
- [42] VASCO.COM. http://www.vasco.com/company/about_vasco/press_room/news_archive/2011/news_diginotar_reports_security_incident.aspx, Sept. 2011.
- [43] WHATSAPP. Encryption Overview. <https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>, Apr. 2016.
- [44] YLONEN, T., AND LONVICK, C. The Secure Shell Transport Layer Protocol. RFC 4253, Jan. 2006. Updated by RFC 6668.
- [45] ZIMMERMANN, P., JOHNSTON, A., AND CALLAS, J. ZRTP: Media Path Key Agreement for Unicast Secure RTP. RFC 6189 (Informational), Apr. 2011.

A Appendix

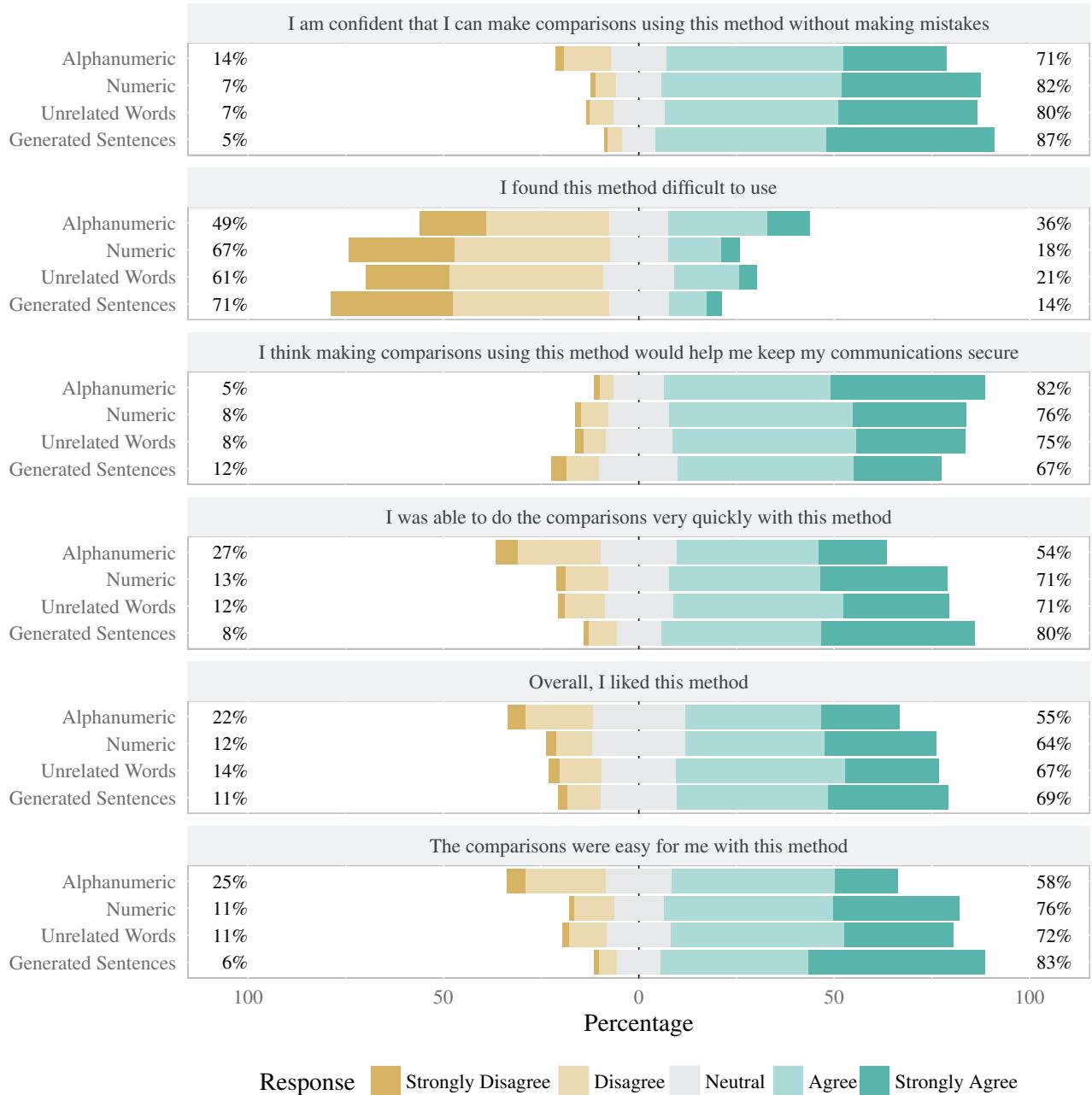


Figure 6: Survey results for all statement ratings

Off-Path TCP Exploits: Global Rate Limit Considered Dangerous

*Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, Lisa M. Marvel[†]
University of California, Riverside, [†]US Army Research Laboratory
{ycao009,zhiyunq,zwang048,tdao006,krish}@cs.ucr.edu, lisa.m.marvel.civ@mail.mil*

Abstract

In this paper, we report a subtle yet serious side channel vulnerability (CVE-2016-5696) introduced in a recent TCP specification. The specification is faithfully implemented in Linux kernel version 3.6 (from 2012) and beyond, and affects a wide range of devices and hosts. In a nutshell, the vulnerability allows a blind off-path attacker to infer if any two arbitrary hosts on the Internet are communicating using a TCP connection. Further, if the connection is present, such an off-path attacker can also infer the TCP sequence numbers in use, from both sides of the connection; this in turn allows the attacker to cause connection termination and perform data injection attacks. We illustrate how the attack can be leveraged to disrupt or degrade the privacy guarantees of an anonymity network such as Tor, and perform web connection hijacking. Through extensive experiments, we show that the attack is fast and reliable. On average, it takes about 40 to 60 seconds to finish and the success rate is 88% to 97%. Finally, we propose changes to both the TCP specification and implementation to eliminate the root cause of the problem.

1 Introduction

TCP and networking stacks have recently been shown to leak various types of information via side channels, to a blind off-path attacker [22, 14, 12, 21, 11, 29, 5]. However, it is generally believed that an adversary cannot easily know whether any two arbitrary hosts on the Internet are communicating using a TCP connection without being on the communication path. It is further believed that such an off-path attacker cannot tamper with or terminate a connection between such arbitrary hosts. In this work, we challenge this belief and demonstrate that it can be broken due to a subtle yet serious side channel vulnerability introduced in the latest TCP specification.

The two most relevant research efforts are the following: 1) In 2012, Qian *et al.*, framed the so called

“TCP sequence number inference attack”, which can be launched by an off-path attacker [22, 23]. However, the attack requires a piece of unprivileged malware to be running on the client to assist the off-path attacker; this greatly limits the scope of the attack. 2) In 2014, Knockel *et al.*, identified a side channel that allows an off-path attacker to count the packets sent between two arbitrary hosts [21]. The limitation is that the proposed attack requires on average, an hour of preparation time and works at the IP layer only (cannot count how many packets are sent over a specific TCP connection).

In this paper, we discover a much more powerful off-path attack that can quickly 1) test whether any two arbitrary hosts on the Internet are communicating using one or more TCP connections (and discover the port numbers associated with such connections); 2) perform TCP sequence number inference which allows the attacker to subsequently, forcibly terminate the connection or inject a malicious payload into the connection. We emphasize that the attack can be carried out by a purely off-path attacker without running malicious code on the communicating client or server. This can have serious implications on the security and privacy of the Internet at large.

The root cause of the vulnerability is the introduction of the *challenge ACK* responses [26] and the global rate limit imposed on certain TCP control packets. The feature is outlined in RFC 5961, which is implemented faithfully in Linux kernel version 3.6 from late 2012. At a very high level, the vulnerability allows an attacker to create contention on a shared resource, i.e., the global rate limit counter on the target system by sending spoofed packets. The attacker can then subsequently observe the effect on the counter changes, measurable through probing packets.

Through extensive experimentation, we demonstrate that the attack is extremely effective and reliable. Given any two arbitrary hosts, it takes only 10 seconds to successfully infer whether they are communicating. If there is a connection, subsequently, it takes also only tens of

seconds to infer the TCP sequence numbers used on the connection. To demonstrate the impact, we perform case studies on a wide range of applications.

The contributions of the paper are the following:

- We discover and report a serious vulnerability unintentionally introduced in the latest TCP specification which is subsequently implemented in the latest Linux kernel.
- We design and implement a powerful attack exploiting the vulnerability to infer 1) whether two hosts are communicating using a TCP connection; 2) the TCP sequence number currently associated with both sides of the connection.
- We provide a thorough analysis and evaluation of the proposed attack. We present case studies to illustrate the attack impact.
- We identify the root cause of the subtle vulnerability and discuss how it can be prevented in the future. We propose changes to the kernel implementation to eliminate or mitigate the side channel.

2 Background

Security was not the primary concern in the design of TCP. There have been many security patches over the years at both the specification and implementation level. Interestingly, most new specifications are well thought out and typically improve security. Unfortunately, as we discover, one of the most recent specifications intended to improve security creates an even more serious vulnerability.

In this section, we first present the threat model that is being addressed in RFC 5961 and how the new specification is supposed to protect against blind in-window attacks. In the next section, we will show that how this specification introduces a new vulnerability.

Threat model: As illustrated in Figure 1, a realistic threat model for TCP is off-path attacks. There are three hosts involved: a victim client, a victim server and an off-path attacker. Any machine might act as the attacker in this model as long as its ISP allows the off-path attacker to send packets to the server with the spoofed IP address of the victim client. Alternatively, as shown in Figure 2, the off-path attacker is able to send packets to the client with the spoofed IP address of the victim server.

Blind in-window attacks: Under the above threat models, the most common attacks considered are “blind in-window attacks” where an off-path attacker sends spoofed TCP packets with guessed sequence numbers in an attempt to achieve DoS or data injection attacks. To succeed in such an attack, it is necessary to first know the target 4-tuple <src IP, dst IP, src port, dst port> of an

ongoing TCP connection between a client and a server¹. Once the correct 4-tuple is known, if the guessed sequence number of the spoofed packet happens to fall in the receive window, (called an in-window sequence number), one can in fact reset or inject *acceptable* malicious data into the connection. To be more precise, an in-window sequence number is one that satisfies the following condition, ($RCV.NXT \leq SEG.SEQ \leq RCV.NXT + RCV.WND$), where $SEG.SEQ$ is the guessed sequence number, $RCV.NXT$ and $RCV.WND$ are the sequence number of the next byte that the receiver expects to receive, and the receive window size, respectively. To carry out a blind attack, one typically needs to blast the entire sequence number space by sending a large sequence of spoofed packets. In this sequence, the sequence number of a packet is larger than that of its predecessor by a window size.

To defend against such attacks, RFC 5961 proposes several modifications on how TCP should process incoming packets. We highlight only the necessary details below.

2.1 Mitigating the Blind Reset Attack using the SYN Bit

An attacker might tear down an existing TCP connection by injecting SYN packets (TCP packets in which the SYN flag is set). This is because a valid SYN packet will cause the receiver to believe that the sender has restarted and thus, the connection should be reset.

In the former (pre-RFC 5961) Linux kernel versions, an incoming SYN packet is processed as follows:

- If the sequence number is outside the valid receive window, the receiver will send an ACK back to sender.
- If the sequence number is in-window, the receiver will reset this connection.

It is obvious that the attacker only needs a single SYN packet with an in-window sequence number to reset an ongoing TCP connection. Instead, RFC 5961 proposes modifications in processing the SYN packets as follows:

- If a receiver sees an incoming SYN packet, regardless of the sequence number, it sends back an ACK (referred to as a challenge ACK) to the sender to confirm the loss of the previous connection.
- If the packet is indeed initiated from the legitimate remote peer, it must have truly lost the previous connection and is now attempting to initiate a new one. Upon receiving the challenge ACK, the remote peer will send a RST packet with the *correct* sequence num-

¹This can be achieved, among other methods, through brute-force attempts.

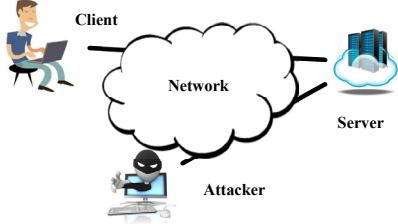


Figure 1: Threat model 1

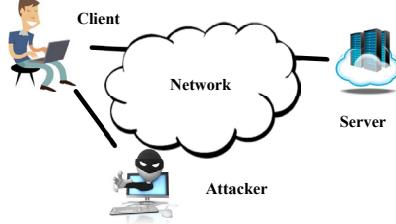


Figure 2: Alternative threat model

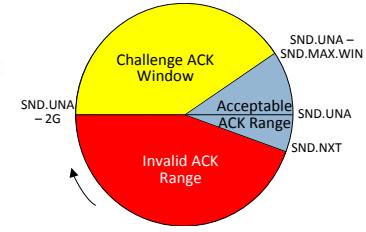


Figure 3: ACK window illustration

ber (derived from the ACK field of the challenge ACK packet) to prove that the previous connection is indeed terminated.

Hence, if the SYN packet is a spoofed one, it can no longer terminate a connection with an in-window sequence number.

2.2 Mitigating the Blind Reset Attack using the RST Bit

An attacker might also tear down the connection by injecting RST packets (TCP packets in which the RST flag is set) into an ongoing TCP connection.

In pre-RFC 5961 Linux kernels, just like in the SYN packet case, a RST packet can terminate a connection successfully as long as its sequence number is in-window. RFC 5961 suggests the following changes:

- If the sequence number is outside the valid receive window, the receiver simply drops the packet. No modifications are proposed for this case.
- If the sequence number *exactly* matches the next expected sequence number ($RCV.NXT$), the connection is reset.
- If the sequence number is in-window but does not exactly match $RCV.NXT$, the receiver must send a challenge ACK packet to the sender, and drop the unacceptable RST packet.

In the final case, if the sender is legitimate, it sends back a RST packet with the correct sequence number (derived from the ACK number in the challenge ACK) to reset the connection. On the other hand, if the RST is spoofed, the challenge ACK packet will not be observable by the off-path attacker. Therefore, the attacker needs to be extremely lucky to be able to succeed — only one out of 2^{32} sequence numbers will be accepted.

2.3 Mitigating the Blind Data Injection

An attacker might corrupt the contents of a transmission by injecting spoofed DATA packets. When a packet arrives, the receiver first checks the sequence number to

make sure it is in-window; in addition, the ACK number will be checked. Pre-RFC 5961, the ACK number is considered valid as long as it falls in the wide range of $[SND.UNA - (2^{31} - 1), SND.NXT]$; this is effectively half of the ACK number space. Here, $SND.UNA$ is the sequence number of the first unacknowledged byte. $SND.NXT$ is the sequence number of the next byte about to be sent.

RFC 5961 suggests a much smaller valid ACK number range of $[SND.UNA - MAX.SND.WND, SND.NXT]$, where $MAX.SND.WND$ is the maximum window size the receiver has ever seen from its peer. This is illustrated in Figure 3. The reasoning is that the only valid ACK numbers are those that are (i) not too old (bytes that are recently sent) and (ii) not too new (receiver cannot ACK bytes that are yet to be sent). The remaining ACK values will be in the range of $[SND.UNA - (2^{31} - 1), SND.UNA - MAX.SND.WND]$, denoted as the *challenge ACK window*. Even though ACK numbers inside this window are still considered invalid, the specification requires the receiver to generate outgoing challenge ACKs in response to packets with such ACK numbers. Overall, this more stringent ACK number check does not eliminate, but helps dramatically reduce the probability that invalid data is successfully injected. Specifically, if the $MAX.SND.WND$ is small (which is typically the case for most connections), then the acceptable ACK window will be much smaller than the half of the ACK number space (as illustrated in Figure 3).

2.4 ACK Throttling

In general, as explained earlier, RFC 5961 enforces a much stricter check on incoming TCP packets; for example, it requires the RST packets to have an exact sequence number to actually reset the connection, whereas a “good enough” in-window value only triggers a challenge ACK. In order to reduce the number of challenge ACK packets that waste CPU and bandwidth resources, an ACK throttling mechanism is also proposed. Specifically, the system administrator can configure the maximum number of challenge ACKs that can be sent out in a given interval (say, 1 second). The RFC clearly states “An implementation SHOULD include an ACK

throttling mechanism to be conservative.” Therefore, the Linux kernel has faithfully implemented this feature by storing the challenge ACK counter in a global variable shared by *all* TCP connections. This approach, unfortunately, creates an undesirable side channel, as will be elaborated. We emphasize that the RFC states that ACK throttling applies to only *challenge ACKs* and not to regular ACKs. This means that the challenge ACK counter is unlikely to be affected by legitimate ACK traffic as the conditions that trigger challenge ACKs are all considered rare or due to attacks.

3 Vulnerability Overview

The Linux kernel first implemented all the features suggested in RFC 5961, in version 3.6 in September 2012. The changes were backported to certain prior distributions as well. The ACK throttling feature is specifically implemented as follows: a global system variable `sysctl_tcp_challenge_ack_limit` was introduced to control the maximum number of challenge ACKs generated per second. It is set to 100 by default. As this limit is shared across all connections (possibly including the connections established with the attacker), the shared state can be exploited as a side channel.

Assuming we follow the threat model in Figure 1, the basic idea is to repeat the following steps: 1) send spoofed packets to the connection under test (with a specific four-tuple), 2) create contention on the global challenge ACK rate limit, i.e., by creating a regular connection from the attacker to the server and intentionally triggering the maximum allowed challenge ACKs per second, and 3) count the actual number of challenge ACKs received on that connection. If this number is less than the system limit, some challenge ACKs must have been sent over the connection under test, as responses to the spoofed packets.

Depending on the types of spoofed packets sent in step 1, the off-path attacker can infer 1) if a connection specified by its four-tuple exists; 2) the next expected sequence number (*RCV.NXT*) on the server (or client); 3) the next expected ACK number (*SND.UNA*) on the server (or client). It is intriguing to realize that the three information leakages are enabled by the three (and only three) conditions that trigger challenge ACKs as described in §2.1, §2.2, and §2.3, respectively.

We elaborate below, the intuition on how the inference can be done in each case.

Connection (four-tuple) inference. Figure 4 shows the sequence of packets that an off-path attacker can send to differentiate between the cases of (i) the presence or (ii) the absence of an ongoing connection. In both cases, the attacker sends the same sequence of packets. Dashed lines represent packets with spoofed IP addresses. In

the figure, the initial SYN-ACK packet is spoofed so that it appears to come from the client. The counter for the number of challenge ACKs that can be issued (100 initially) is tracked and depicted on the timeline of the server.

The hope is that the initial spoofed SYN-ACK packet will hit a correct four-tuple that corresponds to an active connection between the client and the server. In such a case (the left of Figure 4) the server will reply with a challenge ACK² (in accordance with the countermeasure proposed to defend against blind SYN packet injection as described in §2.1). At the same time, this will reduce the global challenge ACK count from 100 to 99. In the case where the spoofed SYN-ACK does not hit a correct four-tuple (on the right of the figure), the server will simply reply with a RST back to the corresponding client (as per TCP standards).

The attacker will then send 100 non-spoofed in-window RST packets to exhaust the challenge ACK count (this behavior is described in §2.2). In the *active connection* case, since the challenge ACK count is 99, the attacker can now observe only 99 challenge ACKs. In the *no connection* case, the attacker can observe 100. The difference in the number of challenge ACKs effectively leaks the information about whether a tested four-tuple corresponds to an active connection or not.

Sequence number inference. Assuming the attacker has already identified a four-tuple that corresponds to an active connection between the client and server, the off-path attacker now needs to guess a valid sequence number that is considered acceptable by the server. Figure 5 shows the sequence of packets that an attacker can send to distinguish between the cases of (i) *in-window* and (ii) *out-of-window sequence number*. In the first case where the spoofed RST packet has an in-window sequence number (but not the next expected sequence number), as per the countermeasure proposed to defend against blind RST packet injection as described in §2.2, a challenge ACK is triggered and this reduces the global challenge ACK count from 100 to 99. In the second case where the sequence number falls outside of the window, no challenge ACK will be generated (the global challenge ACK count remains at 100).

Similar to connection inference, the attacker will now send 100 non-spoofed in-window RST packets to exhaust the challenge ACK count. Once again, based on how many challenge ACKs are received, the attacker can tell if the guessed sequence number in the spoofed RST, is in-window or out-of-window.

ACK number inference. After an in-window sequence number of an active connection is identified, the attacker now will need to guess a valid ACK number that

²The effect is the same as sending a spoofed SYN. However, sending a SYN-ACK is generally more stealthy.

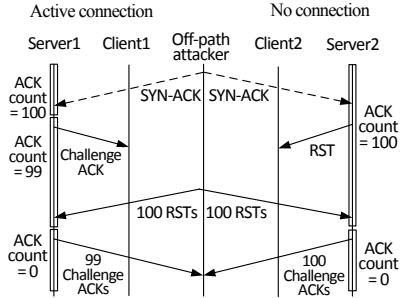


Figure 4: Connection (four-tuple) test

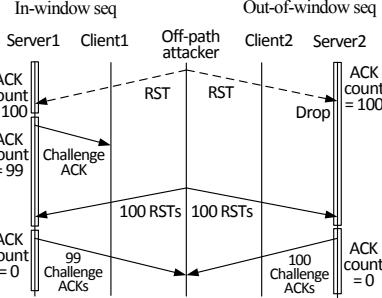


Figure 5: Sequence number test

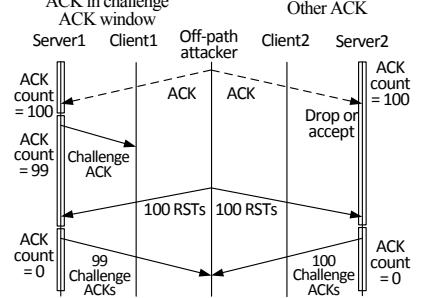


Figure 6: ACK number test

is considered acceptable by the server. Figure 6 shows the sequence of packets that an attacker can send to differentiate the cases of (i) *ACKs in challenge ACK window* and (ii) *other ACK numbers*. In the first case where the spoofed ACK packet has an ACK number in challenge ACK window (but with an in-window sequence number), the server will reply with a challenge ACK, in accordance with the countermeasure proposed to defend against blind data packet injection (as described in §2.3). Following the same procedure as before, an attacker can infer if the guessed ACK number falls in the challenge ACK window. As will be described in §5.2, this helps the attacker to eventually identify the *SND.NXT* on the server.

It is worth noting that once both the sequence number and ACK number acceptable by the server are inferred, an attacker can determine the sequence number and the ACK number acceptable by the client as well. This is because the *RCV.NXT* and *SND.NXT* on the server are basically equivalent to *SND.NXT* and *RCV.NXT* on the client [25, 18]. In practice, if the victim connection has ongoing traffic, the inferred sequence and ACK number may shift as the attack is in progress. We discuss such cases in §6.

An alternative approach for sequence number inference. In some cases a large number of RST packets observed in a short period time may be considered abnormal. Firewalls may even rate limit RST packets on a per-connection basis. In order to alleviate this, one can in fact replace RST packets with ACK packets, which are likely to stay under the radar. As shown in Figure 3, a challenge ACK will be sent when ACK number is in *challenge ACK window* while sequence number is inwindow. Since the *challenge ACK window* space is at least 1/4 of the entire 4G of the ACK number space, one can send 4 packets with ACK numbers 0, 1G, 2G, and 3G respectively and at least one packet will trigger a challenge ACK if the guessed sequence number is inwindow. To understand why the *challenge ACK window* is at least this large, we first point out that the maximum receive window size is 1G with the TCP window scaling

option (RFC 7323), which means that *SND.MAX.WIN* cannot be larger than 1G. Therefore, according to definition of the challenge ACK window described in §2.3, it is at least 1G as well. Given this, every spoofed RST packet sent earlier for sequence number inference is replaced by four ACK packets, which is less efficient but still effective. We have implemented and tested this alternative approach for sequence number inference. However, to simplify the description, we assume the use the original sequence number inference with RST packets in the subsequent sections.

4 Off-Path Connection Reset Attack

In the previous section, we illustrate how the global challenge ACK rate limit can theoretically leak information about an ongoing connection to an off-path attacker. In this section, we focus on how to construct a practical off-path connection reset attack that succeeds when a spoofed RST arrives with a matching sequence number of *RCV.NXT*. This requires an attacker to successfully carry out both *connection (four-tuple) inference* and *sequence number inference*. As will be discussed, to construct a realistic attack, several practical challenges need to be overcome. We assume the threat model to be the one in Figure 1 throughout the section, but the attack works with the alternative threat model (Figure 2) as well.

Goals and constraints. The main goal of the attack is to *quickly* and *reliably* conduct the sequence number inference and use it to reset an ongoing connection. The faster the attack succeeds, the more potent the DoS effect will be. However, the extent of the effect is subject to two practical constraining factors: (i) The bandwidth may be limited between the attacker and the victim (either server or client). (ii) Packet loss may occur between the attacker and victim, especially when they are far away. In this section, we focus only on designing fast probing schemes with given bandwidth constraints and leave the strategy to deal with packet loss to §6.

4.1 Time Synchronization

Challenge: As mentioned in §3, the challenge ACK rate limit is on a per second basis. In other words, the counter for the number of challenge ACK packets that can be issued, gets reset each second. Therefore, it is critical that in each cycle, all the spoofed and non-spoofed packets sent from the attacker arrive within the same 1-second interval, at the server.

One naive solution is that the attacker sends all those packets in a very short period (say, 10 ms), to ensure that the likelihood that they arrive within the same 1-second interval is high. Unfortunately, in practice, this solution does not work well since (i) many factors influence packet delays and thus, the gaps between packet arrival times at the receiver, might be much larger than the gaps in their transmission times, (ii) such bursts of traffic are likely going to experience congestion and packet loss. Thus, it is best for the attacker to synchronize with the clock on the server, so that the attacker can spread the traffic over the 1-second interval, without worrying that some packet arrivals may cross the boundary between two 1-second intervals.

The most common way to synchronize time between two machines is using the Network Time Protocol (NTP). But in practice, the attacker does not know if the server uses NTP, or to what NTP server it connects to; thus, it is not a reliable solution.

Solution: We propose a time synchronization strategy based on the very side channel introduced by the challenge ACK rate limit. The idea is to send more than 200 in-window RST packets spread out evenly in one second and check if we can see more than 100 challenge ACKs; if so, this indicates that we have crossed the boundary between two one second intervals (and have therefore not synced with the server yet). We then adjust the timing for next round of probing (shift it just enough) until we receive exactly the 100 challenge ACKs; in this case, we have succeeded in synchronizing with the server clock.

The reason we choose 200 packets is two-fold: 1) We are able to trigger at most 200 challenge ACKs no matter how many RST packets we send. These 200 challenge ACKs are triggered only when half of the RST packets arrive before the start of a new 1-second interval and half arrive after. 2) By evenly spreading the 200 packets over a 1-second window, i.e., sending one packet every 5ms, allows us to adjust the timing of the next round probing with the finest granularity. Specifically, we show that the time synchronization can be done in at most three rounds of probing in an ideal case (without packet losses).

Round 1: As described before, the attacker sends 200 in-window RST packets to the server evenly spread out over a 1-second window. The attacker then listens and counts the number of received challenge ACK packets.

This value is stored as n_1 . Here, the attacker listens for incoming packets for 2 seconds conservatively, before sending any additional packets to make sure a 1-second interval on the server has elapsed. Note that apart from the 200 RST packets, no other packet is sent to the server in this interval. If n_1 equals 100, it means that all 200 RST packets all arrive in the same 1-second interval on the server, thereby indicating that we have already synchronized with the server. Otherwise, it must be true that $n_1 > 100$, in which case the attacker proceeds to the next round.

Round 2: The attacker waits for 5ms (shifting the start time of the probes by 5ms) and repeats the same process as in the first step. The number of received challenge ACK this time is stored as n_2 . If n_2 equals 100, the synchronization is done. Otherwise, the attacker proceeds to round 3.

Round 3: By comparing n_1 with n_2 , the attacker can determine the final move to be synchronized. Specifically, we provide the following reasoning to support the decision. Assume that in step 1, x RST packets arrive in the first 1-second interval on the server, and y RST packets arrive in the second 1-second interval; note that $x + y = 200$. Similarly, in step 2, there are $(x - 1)$ and $(y + 1)$ RST packets arrive in the first and second 1-second intervals respectively, since in step 2 the attacker time shifts its probes by a period of 1 sub-interval. Thus, $n_1 = \min(x, 100) + \min(y, 100)$ and $n_2 = \min(x - 1, 100) + \min(y + 1, 100)$.

(i) If $n_2 \geq n_1$: Let us assume that $y \geq 100$ and $x \leq 100$; then $n_1 = \min(x, 100) + \min(y, 100) = x + 100$, and $n_2 = \min(x - 1, 100) + \min(y + 1, 100) = (x - 1) + 100 < n_1$, which contradicts the assumption that $n_2 \geq n_1$; thus $y < 100$ and $x > 100$. With these conditions, $n_2 = 100 + (y + 1) = 100 + (200 - x + 1)$, or $(x - 1) = 300 - n_2$. In step 2, $(x - 1)$ RST packets arrive in the first 1-second interval on the server; thus, the attacker has to wait for $(x - 1)$ sub-intervals, i.e., $(300 - n_2) \cdot \frac{1}{200}$ seconds to synchronize her time interval with the server.

(ii) If $n_2 < n_1$: With the same reasoning, the attacker knows that $x < 100$ and $y > 100$. In this case, $n_2 = (x - 1) + 100$; thus, the attacker has to wait $(n_2 - 100)$ sub-intervals, or $\frac{n_2 - 100}{200}$ seconds to synchronize her time interval with the server.

If no packet loss occurs (which is likely due to the small number of packets sent every second), then the three rounds are enough to complete the synchronization process. To handle the rare event that packet loss may occur, we double check that the synchronization was successful by sending another round of 200 RST packets. If it is inconsistent with the previous round, we start over. As will be discussed later, such cases were almost never seen in our experiments.

Algorithm 1: Binary search for source port number

```
1: left = left boundary of the port range
2: right = right boundary of the port range
3: while left < right do
4:   mid = (left + right) / 2
5:   for i = mid to right do
6:     Send a spoofed SYN packet with i as the client port number
7:   end for
8:   Send 100 RST packets on the legitimate connection
9:   Wait until the end of the 1-second interval, count the number of
   received challenge ACK packets
10:  if received ACK packets = 100 then
11:    right = mid - 1
12:  else
13:    left = mid
14:  end if
15: end while
16: return left; //the correct port value
```

4.2 Connection (Four-tuple) Inference

After time synchronization, the attacker can successfully launch subsequent attacks by knowing the boundaries between the 1-second intervals. The first step is “four-tuple inference”, wherein the attacker determines if a connection is established between the client and the server. As mentioned in §2.1, the receiver will send back a challenge ACK (regardless of the sequence number of the packet) when a packet with a SYN flag set, arrives.

In §3, we discussed how this behavior can be exploited to determine whether or not a specific four-tuple is currently active. Basically, for each four-tuple in question, the attacker needs to send a spoofed SYN-ACK packet (a TCP packet in which the SYN and ACK flags are set) with $\langle \text{srcIP} = \text{clientIP}, \text{dstIP} = \text{serverIP}, \text{srcPort} = X, \text{dstPort} = \text{serverPort} \rangle$. The above assumes both the client and server IP addresses are known. In addition, the server port is assumed to be publicly known according to its service type. Therefore, the only unknown is the source port the client uses. The maximum possible port range is $2^{16} = 65536$, and the default range on Linux is only from 32768 to 61000.

A naive approach is to test each port number at a time per second, as depicted in Figure 4, which, in the worst case, requires hours to complete. Therefore, a practical attack requires the attacker to test several port numbers in a second. Let us denote the maximum number of spoofed packets that can be sent in one second by n (constrained by network bandwidth). If n is large, one can search for the port number using a binary-search-like algorithm, the pseudo-code of which is shown in Algorithm 1. Specifically, assuming n is larger than 32767, in the first round the attacker can test the port range from 32768 to 65535 (the most likely half) in a 1-second interval. If the actual port number falls in the range, then the challenge ACK observed by the attacker at the end of the interval will

```
if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq,
                  → TCP_SKB_CB(skb)->end_seq)) {
...
  goto discard;
}
if (th->rst) {
  if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt
      → )
    tcp_reset(sk);
  else
    tcp_send_challenge_ack(sk, skb)
      → ;
  goto discard;
}
```

Figure 7: Logic of handling an incoming packet with RST flag in latest Linux kernels

be 99 (one goes to the victim). If the actual port number *does not* fall in this range, the observed number of challenge ACKs will be 100. In either case, the attacker can narrow down the search space by half and proceed to the next round of search.

An even better strategy is to divide the search space into multiple bins and probe them together in the same round. That way, one can eliminate $\frac{n-1}{n}$ of the search space. A similar multi-bin search strategy is used for sequence number inference in (§4.3).

In cases where n is smaller than 32767 (due to bandwidth constraints), the best the attacker can do is to simply try as many port numbers as possible in each round. The binary search or multi-bin search can be applied later when the search space becomes small enough.

4.3 Sequence Number Inference

As discussed in §2.2, the receiver generates a challenge ACK in response to a RST packet that contains an in-window sequence number which does not match exactly the expected value. The related Linux kernel code is shown in Figure 7; the `tcp_sequence()` function returns true if the sequence number is in-window, and false if it is out-of-window. In the latter case, the packet will simply be dropped. When the sequence number is in-window and the packet has the RST flag set, its sequence number is analyzed further. As we can see, the connection is terminated only when the sequence number matches *RCV.NXT*; otherwise, a challenge ACK is sent.

The main difference between port number inference and sequence number inference is that the attacker does not need to check every possible sequence number to trigger a challenge ACK. Therefore, the attacker can divide the sequence number space into blocks whose sizes are equal to the receive window size, and probe with a guessed sequence number in each block to determine which sequence numbers fall in the receive window. Theoretically, an attacker can apply the same binary

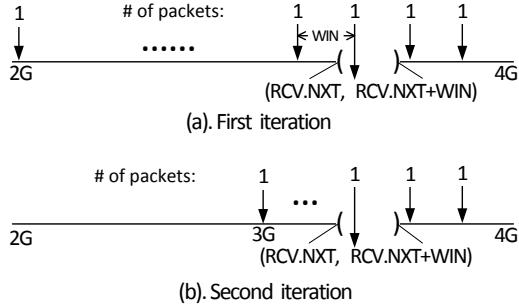


Figure 8: Binary search for sequence number illustration

search algorithm used in connection inference. This process is illustrated in Figure 8. In the first round of probing, the attacker can probe the right half of the sequence number space — (2G, 4G). If any of the spoofed RST packets triggers a challenge ACK, the attacker will observe less than 100 challenge ACKs at the end of the 1-second interval. If there are exactly 100 challenge ACKs observed, it indicates that the receive window is on the left side of the search space. In either case, in the second round, the attacker knows “which half the receive window belongs to.” Let us say that the receive window is in the right half. The attacker would then divide the search space of (2G, 4G) into (2G, 3G) and (3G, 4G). Similar to the first round, only (3G, 4G) needs to be probed in order to determine the part that contains the receive window. This search will eventually stop after 32 rounds exactly (because the sequence number is 32-bit).

However, in practice, the sequence number search space is significantly larger than port number space. Let us consider a receive window size of 12600. This leaves the attacker 340870 possible blocks to search through. If the attacker were to transmit this many packets in one second, the bandwidth requirement would be around 150Mbps, which is extremely high. Likely, the attacker will have to perform a linear search by attempting to search as many blocks as allowed by bandwidth in one second.

Dealing with unknown window sizes: Ideally, the block size should be determined by the window size of the target connection, i.e., the server’s receive window size. In reality, however, an *off-path* attacker cannot observe the window size. If the attacker chooses a smaller window size (compared to the actual window size), the attack will send more packets unnecessarily and take more time. On the other hand, if the guessed value is larger than the actual value, the attacker might miss the correct window of sequence numbers while traversing consecutive blocks. Thus, there is an inherent trade-off between the success rate and the cost incurred (in terms of time and bandwidth) of the attack. Even if the attacker can come up with a correct receive window size at one

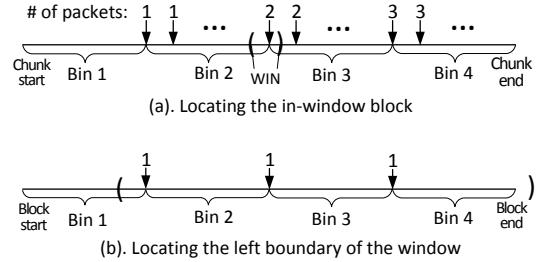


Figure 9: Multi-bin search for sequence number illustration

particular time, the size can change over time.

Our solution is to use a conservative estimate of the window size as the block size in the beginning and update it later given proper feedback. The conservative window size is determined by the initial window size advertised by the server in the SYN-ACK packet. By surveying Alexa top 100 websites, we find that the average initial receive window size is 26703. This window size is the lower bound as the window typically grows after the connection is established. To observe the initial window size, the attacker simply attempts to establish a valid (non-spoofed) TCP connection with the server. This strategy works because a server typically uses the same initial receive window size for all clients. Such a conservative estimate of window size may force the attacker to send more packets, but it at least will guarantee success. We will also discuss how to update the window size dynamically during the search process.

Next, we elaborate the design of sequence number inference:

- **Step 1 – Identify the approximate sequence number range.** Let us assume that the attacker, in n blocks, can send n spoofed packets per second (n is on the order of thousands in our experiments). We call such n consecutive blocks a chunk. The guessed sequence number is always chosen to be the first sequence number within a block. If at the end of the 1-second interval, the attacker observes 100 challenge ACKs, then the attacker proceeds to the next chunk, i.e., the next n consecutive blocks. If the attacker observes less than 100 challenge ACKs, it indicates that the receive window is within the chunk that was just probed. The attack can now proceed to step 2. Note that if the number of observed challenge ACKs is less than 99, it indicates that the initially estimated window size (block size) is too small.

For example, as illustrated in Figure 10(a), if there are two blocks whose beginning sequence numbers are inside the actual receive window, then the number of observed challenge ACKs will be 98; this indicates that the actual window size should be approximately twice the estimated window size (initial block size). We there-

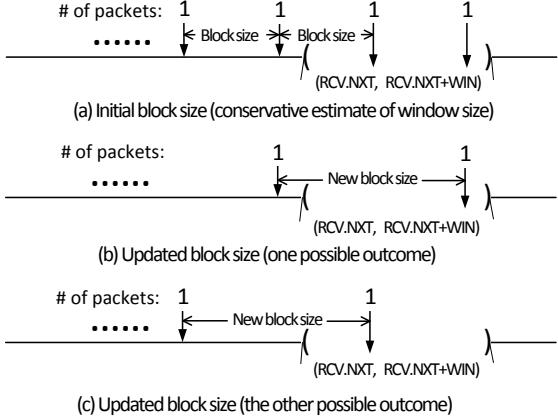


Figure 10: Window size estimate and adjustment

fore update the block size to be twice as much in the subsequent search steps. The two possible outcomes are shown in Figure 10(b) and Figure 10(c).

- **Step 2 – Narrow down the sequence number space to a single block.** From step 1, we know that the receive window is within a chunk. We now further narrow down the search space to an exact block within the chunk. Note that we have now updated the block size so that there will be one and only one block that can trigger challenge ACKs. To locate the exact block, the same binary search strategy outlined in Figure 8 can be used except that the search space now is dramatically reduced after step 1.

The located block has a beginning value which, is an in-window sequence number; therefore, one of the following is true: (i) its beginning value is the correct sequence value; or (ii) the correct sequence value is in its left neighboring block. In the first case, since the sequence number matches the $RCV.NXT$, the spoofed RST packet can already terminate the connection. In the second case, the attacker performs an additional search in the left neighboring block (see Step 3).

- **Step 2 (optimized version) – Identify the correct sequence block using multi-bin search.** With the previous assumption that the attacker can send n spoofed packets per second, with a binary search, the first round requires only $\frac{n}{2}$ packets (as we divide a chunk into two halves initially). The second round requires only $\frac{n}{4}$ packets and so on. As we see, the number of packets sent in each round reduces quickly. This is not an efficient use of the network bandwidth. We show that it is possible to speed up the search process by sending more packets per round (still at most n per round).

The idea is, instead of dividing the search space into two halves in each round, we can divide the space into multiple bins and probe them simultaneously. This is illustrated in Figure 9(a) where 4 bins are present in a chunk. Each bin here holds an equal number of blocks. To determine which bin the receive window falls in, the

attacker sends a different number of spoofed RST packets in each bin. In the example, he sends 1 RST packet per block in the 2nd bin, 2 RST packets per block in the 3rd bin, and 3 RST packets per block in the 4th bin. Since the receive window can fall into one and only one of the bins, the attacker can determine which bin it is in, by observing how many challenge ACKs are received at the end of the 1-second interval. If there are 100 challenge ACKs received, it indicates that the receive window is in the 1st bin (since no RST packets were sent in the 1st bin). Receipt of 99 challenge ACKs indicates that the receive window is in the 2nd bin, etc.

Note that the more bins we have, the faster we can narrow down the sequence number space. However, the number of bins chosen for each round is constrained by n . The larger the n , the more the bins that can be created. The number of bins is also capped at 14, given that the number of spoofed packets may already exhaust the 100 challenge ACK counter in one round ($0 + 1 + 2 + \dots + 13 = 91$).

- **Step 3 – Find the correct sequence number using binary search.** Now we are sure that $RCV.NXT$ is within a specific block, we need to locate its exact value. To achieve the goal, another modified binary search strategy is used here. The observation is that the correct sequence number ($RCV.NXT$) is the highest value in the block, such that any spoofed RST packet with a sequence number less than it will not trigger a challenge ACK packet. It is worth noting that we may not realize which value is the correct sequence number until the connection is terminated, as all the probing packets are RST packets.

- **Step 3 (optimized version) – Find the correct sequence number using multi-bin search.** Similar to the previous multi-bin search, the attacker can divide the single block into many small bins and probe them simultaneously. All bins before the left boundary of the receive window ($RCV.NXT$) will not trigger any challenge ACKs; the ones after will. Thus, in this step attacker only sends one spoofed packet per bin and accumulates all the challenge ACKs received from right to left (See Figure 9). If the attacker sees (100-X) challenge ACKs at the end of the 1-second interval, it indicates that X probed bins are after $RCV.NXT$. In Figure 9, let us say we divide the block into 4 bins. After probing them, the number of observed challenge ACK will be 97 because 2nd, 3rd, and 4th bins turn out to be after $RCV.NXT$. Note that if the observed challenge ACK is 100, it indicates that the correct sequence number is somewhere inside the 4th bin (but not its beginning value).

Similar to the previous multi-bin search, the number of bins chosen for each round is constrained by n . In addition, the number of bins is always capped at 100, as the spoofed packets may exhaust the limit of 100 challenge ACK count.

The RST off-path TCP attack is successfully launched after the above three steps. The exact number of probing rounds depends on the available bandwidth, and will determine the time it takes to finish the attack. We will evaluate this in §7.

5 Off-Path Connection Hijacking Attack

In this section, we discuss how an off-path attacker can hijack an ongoing connection and inject spoofed data. The methodology used to inject data into the client or to the server are similar; thus, without loss of generality, we exemplify the attack targeting the server. First, we describe the challenges that the attacker will need to overcome; subsequently, the entire attack process is described in detail.

5.1 Challenges and Overview

The attacker will experience obstacles that are similar to those associated with launching an off-path reset attack. In addition, the following additional challenges need to be addressed.

Preventing unwanted connection reset. As described in §4.3, the RST packets with in-window sequence numbers are leveraged towards identifying the next expected sequence number on the connection. However, with that process, sending a RST packet with the exact, expected sequence number ($RCV.NXT$) to the server will terminate the TCP connection; this is not the goal of the hijack attack. The challenge is thus, to infer $RCV.NXT$ without causing connection termination.

Identifying both the sequence number and ACK number. In order to trick the server into believing that the injected data is valid, and sent from the server, the attacker needs to know both the correct sequence number ($RCV.NXT$) and the acceptable ACK range on the server side of the connection. The latter is typically a fairly small range as discussed in §2.3.

At a high level, our design of the attack consists of the following steps: First, the attacker finds an in-window sequence number on the server using the techniques described in §4.3. Based on this, the attacker will be able to guess the range of acceptable ACK values that trigger challenge ACKs. The range of these acceptable values (ACK window) can be used to identify the highest acceptable ACK number, i.e., $SND.NXT$, on the server. We will show next that obtaining this ACK number then allows the attacker to infer the exact expected sequence number on the server without resetting the connection.

5.2 Inferring Acceptable ACK Numbers

Assuming an in-window sequence number is already inferred, we now discuss how an attacker can infer the

next ACK number, $SND.UNA$, which is expected by the server. As illustrated in Figure 3, an incoming data packet is accepted if the ACK number is in the range of $[SND.UNA - MAX.SND.WND, SND.NXT]$. If not, the receiver will respond with a challenge ACK packet, if the ACK number is in the range of $[SND.UNA - (2^{31} - 1), SND.UNA - MAX.SND.WND]$; this range is called the *challenge ACK window*. It is obvious that $SND.UNA$ can be computed if one can successfully infer the left boundary of the challenge ACK window, $SND.UNA - (2^{31} - 1)$. This in turn can be found using the following approach.

Step 1: Identify the challenge ACK window position. According to RFC 1323, by using the window scaling option, the maximum receive window size can be extended from 2^{16} to a maximum of $2^{30} = 1G$. Thus, the $MAX.SND.WND$ cannot be larger than 1G. Accordingly, the *challenge ACK window* size is between 1G and 2G, which is one quarter of the entire ACK space size. Because of this, we divide the entire ACK space into 4 bins and probe each bin to check which bin(s) the *challenge ACK window* falls in. In our implementation, we probe the first value of each bin, i.e. 0, 1G, 2G, 3G. We know for certain that either one or two bins can trigger challenge ACK packets. Therefore, we need to send different number of packets for each bin to differentiate the resulting cases. A simple strategy is to send one packet at ACK number 0, two packets at 1G, four packets at 2G, and 8 packets at 3G. For instance, if the number of observed challenge ACKs is 94 (6 missing), then we can infer that both ACK number 1G and 2G have triggered challenge ACKs. If the number of observed challenge ACKs is 96 (4 missing), then only ACK number 2G has triggered challenge ACKs. We can then easily determine the “left-most” bin whose beginning value falls in challenge ACK window.

Step 2: Find the left boundary of the challenge ACK window Now the problem is, given the bin located in the previous step, we need to identify an ACK number in the left neighboring bin, such that it is the “left-most” value (in the circular sense) that can still trigger challenge ACKs. This is a problem that can be solved in a similar way to the last step of sequence number inference using multi-bin search (§4.3).

Finally, when the left boundary of the challenge ACK window ($SND.UNA - (2^{31} - 1)$) is found, an acceptable ACK value ($SND.UNA$) is trivially computed.

5.3 Identify the Exact Sequence Number

To locate $RCV.NXT$ without resetting a connection, we leverage the knowledge learned about the various ACK number ranges. The idea is that, instead of sending spoofed RST packets (which may terminate a connec-

tion), the attacker can send spoofed *data* packets with ACK numbers that fall in the challenge ACK window and thus, intentionally trigger challenge ACKs (if the sequence number is in-window). Combined with the fact no challenge ACK will be triggered if the guessed sequence number is before *RCV.NXT* (considered old packet and dropped), *RCV.NXT* can be located as the “left-most” value that can trigger challenge ACKs. The search process is in fact similar to the last step in sequence number inference except that we now use spoofed data packets.

Now that the attacker knows both the *RCV.NXT* and *SND.UNA* on the server, it is trivial to inject legitimate-looking data packets that will be accepted by the server. Further, it is also trivial to inject legitimate-looking data packets to the client because the *RCV.NXT* on the server is effectively the *SND.UNA* on the client, and the *SND.UNA* is the *RCV.NXT* on the client (assuming no traffic is in flight). In §7.2, we will present a case study on how a web service can be hijacked by a completely blind off-path attacker.

6 Other Practical Considerations

We have fully implemented the attacks described in §4 and §5. In §7, we will evaluate the effectiveness and efficiency of the attacks extensively. In this section, we outline a few practical considerations that need to be handled.

Detecting and handling packet loss. So far, we have assumed that spoofed connections will not incur packet loss and the challenge ACK side channel has no noise. However, in reality, even if the number of packets sent per second is chosen conservatively (well below bandwidth constraints), there is still no guarantee that packet loss will not occur, and a host may legitimately generate challenge ACKs that are not triggered by the attack. They exhibit the same effect to the attacker — the number of observed challenge ACKs will be smaller than expected. In this paper, we call them both packet loss for convenience. We address packet loss based on the two following principles: 1) when in doubt, repeat the probes; 2) add redundancy in the probing scheme to proactively detect packet loss.

In the initial step of the sequence number search, if packet loss occurs, the number of observed challenge ACKs may reduce to 99; the attacker thus, may incorrectly conclude that a chunk that contains the receive window is located. This will affect all subsequent search steps. Therefore, every time when a “plausible” chunk is detected, we repeat the probe on the same chunk. The search will proceed to step 2 only when both rounds return exactly 99 challenge ACKs (no more, no less).

In step 2 and step 3 of the sequence number search,

we add redundancy to actively detect packet loss so that we repeat only the round of probing that experienced packet loss. The idea is similar to using parity bits. In each round, instead of allowing the number of observed challenge ACKs to be any value equal to or below 100, we can construct the probing packets such that only odd number of challenge ACKs will be considered a valid outcome. If an even number of challenge ACKs is received, packet loss must have happened. This strategy can be visualized by referring to Figure 9(a). Instead of sending 1, 2, or 3 packets per block for each bin, we will send 1, 3, and 5 packets per block for each bin. This means that if the receive window falls in 2nd bin, the number of challenge ACKs will be 99; if the receive window is in 3rd bin, the number of challenge ACKs will be 97, etc.

Both schemes are implemented and shown to be very effective in cases where the network conditions between the attacker and the victim are poor.

Moving receive window and challenge ACK window. So far, we have assumed that the connection is relatively idle, and the window does not change while the inference is in progress. This is likely to be the case in many real world scenarios, especially with long-lived connections. One example is the push notification connections on mobile platforms [2]. They are idle most of the time until a new push notification arrives. Even when a connection is not idle at one point, it is likely to become idle at some point and become more susceptible to the attack. Moreover, the traffic activity will mostly be concentrated on either uplink and downlink, rarely both. Typically, downlink traffic dominates; therefore, the attacker targeting at resetting the connection on the server side will experience less difficulty (client’s sequence number increases very slowly). Tor network connections are also candidates as the end-to-end throughput is typically very low.

To support sequence number inference against (slow) moving receive windows, we implement a simple strategy which conducts a brute-force style sequence number guessing. Specifically, once a “left-most” in-window sequence number is inferred (which may become invalid in the next interval due to the ongoing activities), we send 20,000 RST packets with sequence numbers, with offset 1, 2, ..., 20,000 to the valid sequence number. As will be shown in §7.1.2, for low-activity connections, this strategy works well. We leave the exercise to come up with a strategy to target connections with heavier traffic to future work.

Per-connection rate limit. Since the Linux kernel version 4.0 (released in Apr 2015), in addition to the global challenge ACK rate limit, a per-connection rate limit was introduced. The idea is to reduce the impact of potential ACK loops [3] that may occur if client

and server are de-synchronized. Theoretically, the per-connection rate limit provides an isolation between the victim connection and the attacker connection, and the side channel should be eliminated completely. For instance, even if the challenge ACK count limit is reached for the victim connection, it does not affect the limit on the attacker connection at all.

However, interestingly, the per-connection rate limit only applies to SYN packets or packets without any payload. The comment in the Linux kernel states “Data packets without SYNs are not likely part of an ACK loop”, hinting that such packets do not need to be governed by the per-connection rate limit. It is evident that the developers assumed a benign scenario instead of an adversarial one. To get around this restriction, we simply send spoofed packets with a single byte of payload. For the spoofed SYN-ACK packets though, it is impossible to bypass the per-connection rate limit. Unfortunately, upon a closer look at the implementation, when a per-connection challenge ACK is sent out, it is also counted towards the global challenge ACK limit. Therefore, it is still possible to infer that the four-tuple of an ongoing connection has been guessed correctly by observing only 99 challenge ACKs at the end of the 1-second interval. In practice, the per-connection rate limit is 1 packet every 0.5 second, which does allow the attacker to proceed with the binary search approach outlined in §4.2. We have verified experimentally that it does work against the latest Linux kernels with per-connection rate limit.

Configurable maximum challenge ACK count. For simplicity, throughout the paper, we assume the challenge ACK count to be 100, which is the default value. Our test on a variety of Linux operating systems also confirmed the result. However, as proposed in RFC 5961, this value is configurable by a system administrator. According to the specification, the flexibility is provided to allow the tradeoff between resource (bandwidth and CPU) utilization and how fast the system cleans up stale connections. Fortunately, the exact configured value can be inferred quite easily with some simple steps (as long as it is not excessively large). After establishing a legitimate connection to the server, the attacker can send many RST packets, e.g., 1000 packets which is much larger than default value of 100, with in-window sequence values to trigger as many challenge ACKs as possible. The packets are sent in a very short period of time (say, 100 or 200 ms) to increase the likelihood that they end up in the same 1-second interval. The attacker then counts the total number of challenge ACKs returned. Finally, the attacker can wait for a short amount of time and repeat the process one more time to verify the number of received challenge ACK packets is the same; that value would be the actual limit set by the server. Note that this is only a one-time effort for each target.

7 Evaluations

To showcase the effectiveness of our attacks, we next evaluate them in terms of metrics such as success rate and the time to succeed.

7.1 Connection Reset Case Studies

There are two sets of experiments reported in this section viz., where (i) we reset an SSH connection and (ii) perform a Tor connection reset.

Experimental setup. For the SSH experiments, we use a Ubuntu 14.04 host on the University of California - Riverside campus as the victim client. The victim SSH server is one of the instances we create on Amazon EC2 in different geographic locations, worldwide. The attack machine is a Ubuntu 14.04 host in our lab. For the Tor experiments, we target the connection between a Tor relay (set up in our campus) and a random peer relay. Our Tor relay is also a Ubuntu 14.04 host and has been running the service for several months. The attack machine is the same host as the one in the SSH experiments.

In both the SSH and Tor experiments, the attacker attempts to reset the connection on the server end by connecting to it and performing the inference attacks. The diversity of servers and the corresponding network paths help test the robustness of the attack. We assume that the 3-tuple <client IP, server IP, and server port> is known. Further, the attack machine is capable of spoofing the IP address of both the victim client and server.

7.1.1 SSH Connection Reset

Location	Success Rate	Avg # of rounds with loss	Avg % of rounds with loss	BW (pkts)	Time Cost (s)
US West 1	10/10	0	0	5000	48.00
US West 2	9/10	1.0	1.91%	5000	58.00
US East	10/10	0	0	5000	32.00
EU German	9/10	0.3	0.67%	5000	48.00
EU Ireland	10/10	0	0	5000	35.20
Asia 1	10/10	0	0	5000	51.00
Asia 2	9/10	1.7	5.34%	5000	36.67
South America	10/10	0	0	5000	45.70

Table 1: SSH connection reset results

Summary. We run the reset attack against 8 different Amazon EC2 servers in different geographical locations. They are all micro instances set up for our experiments only. We establish a connection from the victim client to each server, and have the attacker perform the off-path connection reset attack. For each server, we repeat the experiment 10 times and report the average. As shown in Table 1, the attack is highly effective: the average success rate is 97% over all runs, with an average time cost of 44.3s. Note that the overall time excludes the time for synchronization (recall §4.1) as it is a one-time effort for a server and can be done a priori. The bandwidth cost

here is 5000 spoofed packets per second, which translates to 4Mbps. Note that the probing scheme has already built in packet loss detection using “parity bits” as described in §6. To show that the packet loss detection scheme works, we report the number of rounds and the percentage of rounds on average, when packet loss is detected. For instance, even when packet loss between the attack node and “Asia 2” server is frequent, we still manage to succeed 9 times out of 10.

Failures may still occur since the detection scheme is rudimentary and may fail to detect packet loss. In some cases, the failure can also be the result of the attacker and server becoming out-of-sync due to network delay variance. The success rate can be further improved by adding more redundancy and using better error detection schemes. However, we argue that the current success rate is already good enough to carry out effective DoS attacks.

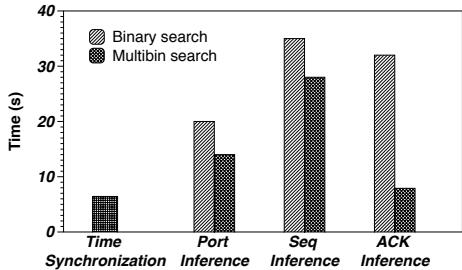


Figure 11: Time breakdown

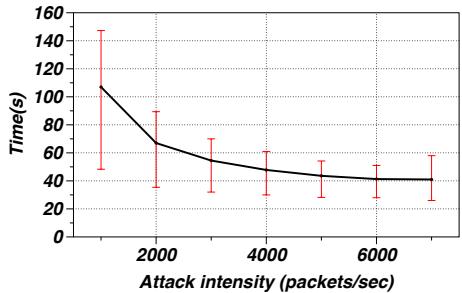


Figure 12: Attack intensity impact on time to succeed

Time breakdown. To understand where the time is spent in our attacks, we conduct another benchmark experiment against one of the SSH servers with both sequence number and ACK number inference. As shown in Figure 11, we break down the time spent into time synchronization and the three search phases of port number inference, sequence number inference, and ACK number inference. We also compare the optimized multi-bin search versus the regular binary search in each phase. Time synchronization takes around 7 seconds (optimization is not applicable). As discussed, it is only a one-time

effort and therefore not on the “critical path”. We see that with the optimized multi-bin search, the time spent on port search is fairly short (around 14 seconds). The time spent on sequence number search takes the most time due to the fact that the sequence number space is much larger. The time spent on ACK number inference is also fairly short (around 8 seconds) due to the fact that the challenge ACK window is extremely large and easy to locate.

Compared to the results with binary search, we see that the optimized multi-bin search has greatly improved the search speed by more than 30 seconds overall. This is due to the fact that binary search significantly underutilizes the bandwidth resources and significantly increases the number of rounds of probes. The reason why the sequence number search does not benefit as much is because most of the time is spent on the initial linear search of the huge sequence number space. This step cannot be optimized with the multi-bin search.

Attack intensity vs. Time to succeed. Using the same experimental setup as before, we vary the attack intensity, i.e., the number of packets sent per second and show how this affects the time it takes to succeed. As shown in Figure 12, we plot the average, min, and max time to successfully conduct sequence number inference only (reset attack), as well as with the ACK number inference added (hijacking attack). Clearly, the higher the attack intensity the faster the attack. When the intensity is only \approx 512 Kbps (1000 packets per second), the time to succeed is over 100 seconds, on average. When the intensity is \approx 4 Mbps, (5000 packets per second), the average time reduces to \approx 50 seconds for hijacking and only 30 seconds for reset. Note that an intensity $>$ 4 Mbps does not substantially improve the time to succeed because we begin to observe more packet losses, which cause additional rounds of probing. Of course, this is experienced on the specific network environment between the attack host and the server, which could differ elsewhere; if the network conditions are even better, the time to succeed can be further improved.

7.1.2 Tor Connection Reset

Node	Target	Success Rate	Avg # of rounds with loss	Avg % of rounds with loss	BW (pkts)	Time Cost(s)
62.210.x.x	FR	8/10	1.9	4.58%	4000	46.36
89.163.x.x	DE	9/10	4.0	7.97%	4000	49.08
178.62.x.x	GB	7/10	3.2	4.20%	4000	53.00
198.27.x.x	NA	10/10	0.8	1.45%	4000	59.86
192.150.x.x	NL	8/10	4.1	5.64%	4000	68.03
62.210.x.x	FR	6/10	2.5	5.85%	4000	49.57
89.163.x.x	DE	8/10	1.7	3.06%	4000	52.51
178.62.x.x	GB	8/10	6.0	8.15%	4000	78.35
198.27.x.x	NA	7/10	2.1	3.64%	4000	72.49
192.150.x.x	NL	6/10	5.5	7.14%	4000	79.42

Table 2: Tor connection reset results (first half under browsing traffic and second half under file downloading traffic)

To conduct a realistic experiment, we use a Tor relay set up in our campus and have a user using it as an entry relay. The entry relay establishes connections with an arbitrary middle relay (anywhere in the world). For ethical reasons, we do not perform attacks against arbitrary relay nodes that are not connected to our node.

To understand how the attack performs against mostly idle connections, we test it against connections between our own Tor relay and 40 other Tor relays throughout the world. The attack node has to connect to these Tor relays that are far away to perform attacks. In each case, we repeat the reset experiment 10 times. First, we discover that 16 of them do not appear vulnerable to the side channel attacks, even though they appear to be Linux hosts. We suspect that this is because of certain firewalls that drop our spoofed packets. For the remaining 24 hosts, the average success rate is 88.8% and the average time to succeed is 51.1s. We find these results to be slightly worse than those in the SSH experiments because of higher packet loss rates.

In addition, we pick 5 random relays and simulate background traffic with browsing and file downloading, and conduct the same experiment as above. Here, to deal with moving windows, we use the simple brute-force strategy described in § 6. The results are shown in Table 2. The average success rate is now down to 77% and the average time to succeed is 60.9s. Upon further inspection, the increased failure rate is exactly due to the moving window problem i.e., it interferes with the sequence number search. Nevertheless, we think the result is acceptable as we have not designed a robust solution specifically for dealing with a moving window (this is left for future work).

In general, we believe that a DoS attack against Tor connections can have a devastating impact on both the availability of the service as a whole and the privacy guarantees that it can provide. The default policy in Tor is that if a connection is down between two relay nodes, say a middle relay and an exit relay, the middle relay will pick a different exit relay to establish the next connection. If an attacker can dictate which connections are down (via reset attacks), then the attacker can potentially force the use of certain exit relays.

7.2 TCP Hijacking Case Study

Our attack does not require any assistance from client-side or server-side malware or puppet (which are required in prior studies [23, 14]). Therefore, our target is any long-lived TCP connection that does not use SSL/TLS. There are several attractive targets: video, advertisements, news, and Internet chat rooms (e.g., IRC). Depending on the implementation, one can envision the following possibilities: 1) the client periodically initiates a request and asks for responses, or 2) the server proac-

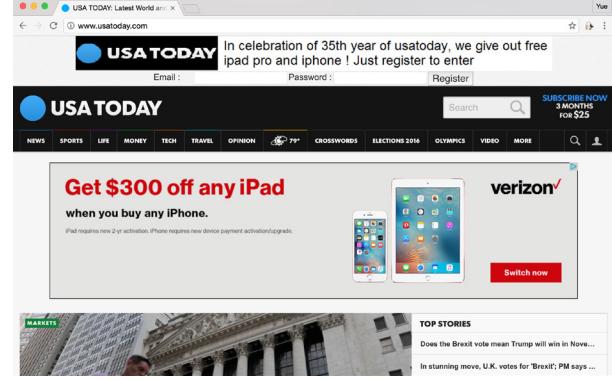


Figure 13: USAToday screenshot with phishing registration window

tively pushes notification messages. In both cases, our attack can inject malicious messages to the client and induce a variety of classic attacks such as phishing or cross-site scripting.

Here, we pick a news website www.usatoday.com which has a long-lived TCP connection that periodically retrieves news updates every 30 seconds. This gives ample idle time for our sequence number and ACK number inference. The attacker machine and the victim client are Ubuntu 14.04 hosts in our lab (as in the other case studies). Once the numbers are inferred, we perform a de-synchronization attack [4] by sending a spoofed request to the server that will force it to send a response to the client. Since the request was never sent by the client, it will not accept the response as the response packet contains an invalid ACK number (acknowledging data that have not been sent). Later, when the client itself initiates a real request, the server would no longer accept it as the packet is considered to be data with an old sequence number. Now that the client and server become de-synchronized, the attacker no longer needs to worry about a race condition where the response to the victim client is sent back by the server first. During all this, the attacker simply sends spoofed responses periodically every few seconds with ACK numbers properly acknowledging the client’s requests. If such spoofed responses arrive before the client sends a request, they will simply be dropped without any adverse effect (because the ACK numbers are acknowledging data that has not been transmitted yet).

We implement the attack end to end, and successfully hijack the connection and inject a phishing registration window to solicit email and passwords at the top of the webpage as shown in Figure 13. We repeat the experiment 10 times and summarize our results in Table 3. The attack first infers sequence and ACK numbers before injecting the malicious payload. Success rate 2 quantifies the rate of inferring the sequence and ACK numbers correctly. However, USAToday occasionally switches the

Success rate 1	Success rate 2	Avg # of rounds with loss	Avg % of rounds with loss	BW (pkts)	Time (s)	Cost
7/10	9/10	2.22	3.63%	5000	81.05	

Success rate 1 = success rate of injecting the phishing registration window
 Success rate 2 = success rate of inferring the correct sequence and ACK number

Table 3: USAToday injection results

HTTP request from one type to another and therefore the injected payload will not match the request. Success rate 1 quantifies the rate of injecting the response that matches the request, which is strictly lower than success rate 2, but is still reasonable in our experiments. In addition, the time to succeed is longer than in the case of SSH and Tor experiments mostly because of the extra steps of ACK number inference and data injection.

8 Discussion and Defenses

Vulnerabilities in other OSes: We examine if the studied vulnerability exist in the latest Windows and FreeBSD OSes (The latter TCP stack is also used by Mac OS X). In brief, these OSes are not vulnerable to the attack. First of all, neither Windows nor FreeBSD has implemented all three conditions that trigger challenge ACKs according to RFC 5961. More importantly, the ACK throttling is not found for Windows or MAC OS X. Ironically, not implementing the RFC fully, in fact is safer in this case.

Defenses. As highlighted earlier, the root cause of all the attacks described is the side channel associated with the global challenge ACK count. This side channel can leak various types of information about an ongoing TCP connection. In general, as asserted in previous studies [21], network protocols are not designed rigorously to guarantee the non-interference property. In our study, we discover that the design and implementation of RFC 5961 has actually introduced an information flow that leaks TCP connection state through the shared challenge ACK counter, and is highly exploitable.

The best defense strategy is to eliminate the side channel (the global challenge ACK count) altogether. One can still enable the per-connection rate limit as long as each connection has a completely separate counter that does not interfere with those of other connections. The downside of this strategy is that if the number of connections in a system increases, the aggregate challenge ACK count can go up without any bound. There is currently no evidence to suggest that this worst case scenario is likely to ever happen. However, if one is really concerned about wasting resources on sending challenge ACKs, we suggest a second solution which is adding noise to the channel. This is a common defense strategy in mitigating side channel attacks [10, 27]. Specifically, instead of having a fixed global challenge ACK count of 100 in all intervals, we can add random values (either positive or nega-

tive) for each interval. This will essentially confuse the attacker during the search process. In fact, even if the attacker repeats the probe many times, the result will always differ over time. To ensure that the added randomness is theoretically sound, one can even apply differential privacy to systematically introduce noise, as was done recently in [28]. We leave the design of the exact scheme to add randomness to future work. We also plan to propose the defenses to the Linux community.

9 Related Work

Previous work on off-path TCP sequence number inference heavily relies on executing malicious code on the client side [22, 23, 14, 16, 17, 1], either in the form of malware [22, 23] or malicious javascript [14, 16, 17]. They share the same scheme of “guess-then-check” based on some side channels observable by the malicious code on the client side. They include OS packet counters [22, 23, 9], global IPID [14, 1], and HTTP responses [16]. In contrast, our off-path TCP attack eliminates the requirement completely, which makes the attack much more dangerous. The only prior study that shares the same threat model is the one reported by lkm in phrack magazine in 2007 [1]. The authors exploit the well-known global IPID side channel on Windows hosts to perform such attacks. Unfortunately, the IPID side channel is extremely noisy and the attack can take close to 20 minutes to succeed, as reported by the authors. Furthermore, as reported in [14], the success rate of such an attack is very low, unless the attacker has a low latency to the victim (e.g., on the same LAN). In comparison, our newly reported attack finishes much faster and is significantly more reliable.

Besides the TCP sequence number, it has been shown that other types of information can be inferred by an off-path or blind attacker [12, 21, 11, 29, 5, 15]. For instance, Ensafi *et al.* [12] show that, by leveraging the SYN cache and RST rate limit on FreeBSD, one can infer if a port is open on a target host through bouncing scans off of a “zombie” FreeBSD host. Knockel *et al.* [21] demonstrate the use of a new per-destination IPID side channel that can leak the number of packets sent between two arbitrary hosts on several major operating systems with a bootstrapping time of an hour on average. Alexander *et al.* [5] can infer the RTT between two arbitrary hosts with reasonable accuracy within minutes. Gilad *et al.* [15] are also able to infer if two hosts have established a TCP connection identified by a specific four-tuple, by utilizing the same noisy global IPID side channel. Compared to the newly discovered side channel, it has the following limitations: 1) requires the presence of stateful firewall or NAT which may not be universally present; 2) has a low success rate even when the

tests are repeated multiple times (e.g., for more than a minute). Utilizing the new side channel, we can do this much faster.

Many of the side channels can be abused and cause unwanted information leakage. However, in some cases, they can also be used legitimately for network measurements. For instance, the global IPID side channel has been used to infer a network’s port blocking policy [24]. The same side channel has also been used to count how many hosts are behind a NAT [6]. In addition, even though commonly considered a vulnerability, ISPs that allow IP spoofing are still prevalent according to the latest reports in 2009 [7] and 2013 [8]. Further, very recently, IP spoofing has also been used in legitimate applications such as reverse traceroute [20], detecting Inter-domain Path changes [19], and detecting routing policy violations [13].

10 Conclusions

To conclude, we have discovered a subtle yet critical flaw in the design and implementation of TCP. The flaw manifests as a side channel that affects all Linux kernel versions 3.6 and beyond and may possibly be replicated in other operating systems if left unnoticed. We show that the flaw allows a variety of powerful blind off-path TCP attacks. Finally, we propose changes to the design and implementation of TCP’s global rate limit to prevent or mitigate the side channel.

Acknowledgement

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. The work is also supported by National Science Foundation under Grant #1464410.

References

- [1] Blind TCP/IP Hijacking is Still Alive. <http://phrack.org/issues/64/13.html>.
- [2] Cloud Messaging. <https://developers.google.com/cloud-messaging/>.
- [3] [tcpm] mitigating TCP ACK loop (“ACK storm”) DoS attacks. <https://www.ietf.org/mail-archive-archive/web/tcpm/current/msg09450.html>.
- [4] ABRAMOV, R., AND HERZBERG, A. Tcp ack storm dos attacks. *Journal Computers and Security* (2013).
- [5] ALEXANDER, G., AND CRANDALL, J. R. Off-Path Round Trip Time Measurement via TCP/IP Side Channels. In *INFOCOM* (2015).
- [6] BELLOVIN, S. M. A Technique for Counting Naked Hosts. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurement* (2002).
- [7] BEVERLY, R., BERGER, A., HYUN, Y., AND K CLAFFY. Understanding the Efficacy of Deployed Internet Source Address Validation Filtering. In *Proc. ACM SIGCOMM IMC* (2009).
- [8] BEVERLY, R., KOGA, R., AND K CLAFFY. Initial Longitudinal Analysis of IP Source Spoofing Capability on the Internet. In *Internet Society Article* (2013).
- [9] CHEN, Q. A., QIAN, Z., JIA, Y. J., SHAO, Y., AND MAO, Z. M. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *CCS* (2015).
- [10] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy* (2010).
- [11] ENSAFI, R., KNOCKEL, J., ALEXANDER, G., AND CRANDALL, J. R. Detecting Intentional Packet Drops on the Internet via TCP/IP Side Channels. In *PAM* (2014).
- [12] ENSAFI, R., PARK, J. C., KAPUR, D., AND CRANDALL, J. R. Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks using Model Checking. In *USENIX Security* (2010).
- [13] FLACH, T., KATZ-BASSETT, E., AND GOVINDAN, R. Quantifying Violations of Destination-based Forwarding on the Internet. In *IMC* (2012).
- [14] GILAD, Y., AND HERZBERG, A. Off-Path Attacking the Web. In *USENIX WOOT* (2012).
- [15] GILAD, Y., AND HERZBERG, A. Spying in the Dark: TCP and Tor Traffic Analysis. In *PETS* (2012).

- [16] GILAD, Y., AND HERZBERG, A. When tolerance causes weakness: the case of injection-friendly browsers. In *WWW* (2013).
- [17] GILAD, Y., HERZBERG, A., AND SHULMAN, H. Off-Path Hacking: The Illusion of Challenge-Response Authentication. *Security Privacy, IEEE* (2014).
- [18] HAN, B., AND BILLINGTON, J. Termination properties of TCP's connection management procedures. In *ICATPN* (2005).
- [19] JAVED, U., CUNHA, I., CHOHNES, D., KATZ-BASSETT, E., ANDERSON, T., AND KRISHNAMURTHY, A. Poiroot: Investigating the root cause of interdomain path changes. In *SIGCOMM* (2013).
- [20] KATZ-BASSETT, E., MADHYASTHA, H. V., ADHIKARI, V. K., SCOTT, C., SHERRY, J., VAN WESEP, P., ANDERSON, T., AND KRISHNAMURTHY, A. Reverse Traceroute. In *NSDI* (2010).
- [21] KNOCKEL, J., AND CRANDALL, J. R. Counting Packets Sent Between Arbitrary Internet Hosts. In *FOCI* (2014).
- [22] QIAN, Z., AND MAO, Z. M. Off-Path TCP Sequence Number Inference Attack – How Firewall Middleboxes Reduce Security. In *IEEE Symposium on Security and Privacy* (2012).
- [23] QIAN, Z., MAO, Z. M., AND XIE, Y. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *CCS* (2012).
- [24] QIAN, Z., MAO, Z. M., XIE, Y., AND YU, F. Investigation of Triangular Spamming: A Stealthy and Efficient Spamming Technique. In *Proc. of IEEE Security and Privacy* (2010).
- [25] R. BRADEN, ED. Requirements for Internet Hosts - Communication Layers. rfc 1122, 1989.
- [26] RAMAIAH, ANANTHA AND STEWART, R AND DALAL, MITESH. Improving TCP's Robustness to Blind In-Window Attacks. rfc5961, 2010.
- [27] SONG, D. X., WAGNER, D., AND TIAN, X. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security* (2001).
- [28] XIAO, Q., REITER, M. K., AND ZHANG, Y. Mitigating storage side channels using statistical privacy mechanisms. In *CCS* (2015).
- [29] ZHANG, X., KNOCKEL, J., AND CRANDALL, J. R. Original SYN: Finding Machines Hidden Behind Firewalls. In *INFOCOM* (2015).

Website-Targeted False Content Injection by Network Operators

Gabi Nakibly^{1,3}, Jaime Schcolnik², and Yossi Rubin¹

¹Rafael – Advanced Defense Systems, Haifa, Israel

²Computer Science Department, Interdisciplinary Center, Herzliya, Israel

³Computer Science Department, Technion, Haifa, Israel

Abstract

It is known that some network operators inject false content into users' network traffic. Yet all previous works that investigate this practice focus on edge ISPs (Internet Service Providers), namely, those that provide Internet access to end users. Edge ISPs that inject false content affect their customers only. However, in this work we show that not only edge ISPs may inject false content, but also non-edge network operators. These operators can potentially alter the traffic of *all* Internet users who visit predetermined websites. We expose this practice by inspecting a large amount of traffic originating from several networks. Our study is based on the observation that the forged traffic is injected in an out-of-band manner: the network operators do not update the network packets in-path, but rather send the forged packets *without* dropping the legitimate ones. This creates a race between the forged and the legitimate packets as they arrive to the end user. This race can be identified and analyzed. Our analysis shows that the main purpose of content injection is to increase the network operators' revenue by inserting advertisements to websites. Nonetheless, surprisingly, we have also observed numerous cases of injected malicious content. We publish representative samples of the injections to facilitate continued analysis of this practice by the security community.

1 Introduction

Over the last few years there have been numerous reports of ISPs that alter or proxy their customers' traffic, including, for example, CMA Communications in 2013 [7], Comcast in 2012 [19], Mediacom in 2011 [10], WOW! in 2008 [31], and Rogers in 2007 [36]. Moreover, several extensive studies have brought the details of this practice to light [20, 34, 28, 39]. The main motivations of ISPs to alter traffic are to facilitate caching, inject advertisements into DNS and HTTP error messages, and compress or

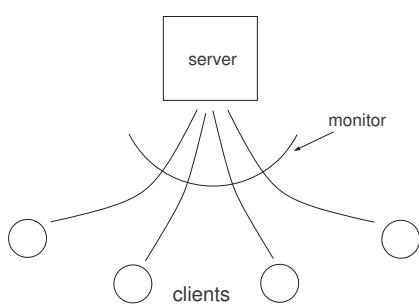
transcode content.

All of these reports and studies found that these traffic alterations were carried out exclusively by *edge ISPs*, namely, retail ISPs that sell Internet access directly to end customers, and are their "first hop" to the Internet. This finding stems from the server-centric approach the above studies have taken. In this approach, one or a handful of servers are deployed to deliver specific content to users, after which a large number of clients are solicited to fetch that content from the servers. Finally, an agent on the clients – usually a JavaScript delivered by the server itself – looks for deviations between the content delivered by the server and that displayed to the user. Figure 1(a) illustrates the traffic monitored in this server-centric approach.

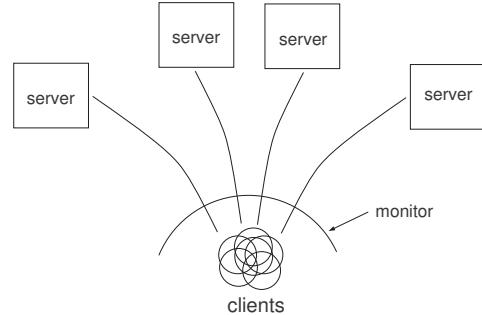
Such an approach can be used to inspect the traffic of many clients from diverse geographies who are served by different edge ISPs. The main disadvantage of this approach is that the content fetched by the clients is very specific. All clients fetch the same content from the same web servers. This allows only the detection of network entities that aim to modify all of the Internet traffic¹ of a predetermined set of users and are generally oblivious to the actual content delivered to the user. Such entities indeed tend to be edge ISPs that target only the traffic of their customers.

In this work we show that the above approach misses a substantial portion of the on-path entities that modify traffic on the Internet. Using extensive observations over a period of several weeks, we analyzed petabits of Internet traffic carrying varied content delivered by servers having over 1.5 million distinct IP addresses. We newly reveal several network operators that modify traffic not limited to a specific set of users. Such network operators alter Internet traffic on the basis of its content, primarily by the website a user visits. The traffic of *every* Internet

¹In some cases these network entities modify all internet traffic originating from very popular websites such as google.com, apple.com, and bing.com or all Internet traffic originating from .com.

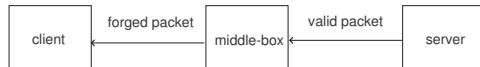


(a) Depiction of monitored traffic in the server-centric approach (of past works). One server with specific content serves many clients in many edge networks.

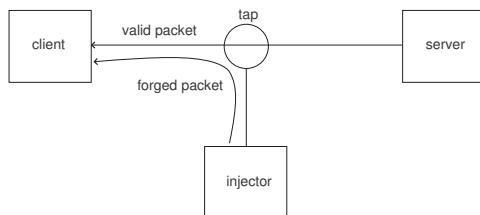


(b) Depiction of monitored traffic in the client-centric approach (of the current work). Many servers with varied content serve many clients in a few edge networks.

Figure 1: Server-centric approach versus client-centric approach to monitoring traffic. The lines between clients and servers illustrate the monitored traffic.



(a) In-band alteration of packet by a middle-box. Only a single packet arrives at the client.



(b) Out-of-band injection of a forged packet. Two packets arrive at the client.

Figure 2: In-band versus out-of-band alteration of content

user that traverses these network operators is susceptible to alteration. This is in contrast to the case of edge ISPs that alter the traffic of their customers only. Although a primary focus of these network operators is to inject advertisements into web pages, we also identified injections of malicious content.

Our analysis is based on the observation that network operators alter packets *out-of-band*: all traffic is passively monitored, and when the content of a packet needs to be altered, a forged packet is injected into the connection between the server and the client. The forged packet poses as the valid packet. If the forged packet arrives at the client before the valid one, the client will accept the forged packet and discard the valid one. Such an approach has considerable advantages to the network operators since it does not introduce new points of failure to their traffic processing and there is no potential for a per-

formance bottleneck. Figure 2 illustrates the differences between in-band alteration of traffic and out-of-band alteration. Note that both in-band and out-of-band traffic alteration is possible only on unprotected traffic, e.g., traffic that is not carried by TLS [12] or authenticated using TCP authentication [32].

The out-of-band operation has a crucial characteristic that enables our analysis: the client receives two packets – the forged one and the valid one – that claim to be the same response from the server. However, they carry different content. This characteristic allows us to detect traffic alteration events while monitoring the traffic at the edge network. We can thus monitor and analyze traffic in a client-centric manner in which the traffic is not destined to a specific set of servers but to all servers contacted by the users at the edge network. Figure 1(b) illustrates the traffic monitored in our work. In this paper we specifically focus our analysis on alteration of *web* traffic, i.e., HTTP traffic over port 80.

An example of out-of-band injection To illustrate how content is altered using out-of-band injection, we describe in the following one of the injections we identified during our observations. In this example the user’s browser sends the following HTTP GET request to cnzz.com (a Chinese company that collects users’ statistics):

```
GET /core.php?show=pic&t=z HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64)
Host: c.cnzz.com
Accept-Encoding: gzip
Referer: http://tfkp.com/
```

In response the user receives two TCP segments having the same value in the sequence number field. The segments include different HTTP responses. One segment

carries the legitimate HTTP response that includes the requested resource (a JavaScript code) from cnzz.com:

```
HTTP/1.1 200 OK
Server: Tengine
Content-Type: application/javascript
Content-Length: 762
Connection: keep-alive
Date: Tue, 07 Jul 2015 04:54:08 GMT
Last-Modified: Tue, 07 Jul 2015 04:54:08 GMT
Expires: Tue, 07 Jul 2015 05:09:08 GMT

!function(){var p,q,r,a=encodeURIComponent,c=...
```

The other segment includes a forged response that directs the user via a 302 status code to a different URL that points to a different JavaScript code:

```
HTTP/1.1 302 Found
Connection: close
Content-Length: 0
Location: http://adcpc.899j.com/google/google.js
```

Our analysis shows that this JavaScript redirects the user through a series of affiliate ad networks ending with Google’s ad network, which serves the user an ad. In this injection event the forged segment arrived before the legitimate one, which means that the user sees the injected ad instead of the original content.

Relation to censorship Website-targeted false content injection is similar in some ways to content blocking for the purpose of state-sponsored censorship. There is a substantial body of work that studies the mechanisms and characteristics of censorship worldwide [33, 37, 22, 9]. In many cases this blocking of content is also website-targeted. Moreover, blocking is often done by injecting false traffic segments, which in some cases is done out-of-band [33, 11, 8]. In contrast to previous works on censorship, in this work we study the practice of false content injection by commercial network operators, rather than state entities. Such injections primarily serve financial gains rather than political agenda, with the goal of *altering* the web content rather than blocking it. In this work we study and analyze the practice of financially-motivated false content injection by network operators. In Section 7 we discuss in more detail related work on censorship. During this work we observed numerous occurrences of censorship-aimed injections. We do not report on them in this paper.

Our contributions can be summarized as follows:

1. The observation that network operators inject false web content out-of-band.
2. Investigation of the identities of network operators that practice website-targeted content injection.

3. Thorough analysis of the characteristics of the injections and the purpose of the injecting operators.

The paper’s structure is as follows. In Section 2 we present technical background pertaining to injection of forged TCP and HTTP packets. Section 3 details our methodology for monitoring web traffic and identifying injections of forged packets. Section 4 details the sources of traffic we monitored. In Section 5 we present our analysis of the injection events and our investigation as to the identities of the network operators behind them. Section 6 proposes effective and efficient client-side mitigation measures. Section 7 discusses related work and Section 8 concludes the paper.

2 Background

2.1 Out-of-band TCP Injection

A TCP [27] connection between two end nodes offers reliable and ordered delivery of byte streams. To facilitate this service, every sent byte is designated a sequence number. Each TCP segment carries a Sequence Number field that indicates the sequence number of the first data byte carried by the segment. The following data bytes in the segment are numbered consecutively. A third party that wishes to send a forged TCP segment as part of an existing TCP connection must correctly set the connection’s 4-tuple in the IP and TCP header, i.e., the source’s port number and IP address as well as those of the destination. In addition, for the forged segment to be fully accepted by the receiver, the sequence numbers of the forged data bytes must fully reside within the receiver’s TCP window. Forging such a TCP segment is trivial for an on-path third party, since it can eavesdrop on the valid segments of the connection and discover the 4-tuple of the connection as well as the valid sequence number.

In some circumstances an injected TCP segment may trigger an undesirable “Ack storm”. An “Ack storm” occurs when the injected segment causes the receiver to send an acknowledgment for data bytes having sequence numbers that were not yet sent by the peer. Appendix A details how an “Ack storm” is formed. Nonetheless, as long as the injecting third party ensures that the injected TCP segment is no larger than the valid TCP segment sent by the peer, no “ACK storm” will be triggered. If this is not the case, the injector could send a TCP reset right after the injection in order to forcibly close the connection. This will also eliminate the possibility of an “Ack storm”. The latter option is used only if the connection is expected to close right after the valid response is received. Indeed, in all our observations either of these alternatives took place and no “Ack storms” were observed.

Nonetheless, the fact that the injected TCP segment aims to displace an already sent or soon to be sent valid TCP segment poses a different obstacle for the injecting third party. According to the TCP specification [27], the first data byte received for a given sequence number is accepted. A subsequent data byte having the same sequence number is always discarded as a duplicate regardless of its value. Thus, the injected segment must arrive at the receiver *before* the valid TCP segment in order to be accepted. Note that the TCP specification does not consider the receipt of bytes with duplicate sequence numbers as an error but rather as a superfluous retransmission.

2.2 HTTP Injection

In this work we focus in particular on the injection of false HTTP responses received by a web client. HTTP [15] is a stateless client-server protocol that uses TCP as its transport. An HTTP exchange begins by a client sending an HTTP request, usually to retrieve a resource indicated by a URI included in the request. After processing the request the server sends an HTTP response with a status code. The status codes we later refer to in this paper are:

- 200 (Successful): The request was successfully received, understood, and accepted. Responses of this type will usually contain the requested resource.
- 302 (Redirection): The requested resource resides temporarily under a different URI. Responses of this type include a Location header field containing the different URI.

An HTTP client will receive only one HTTP response for a given request even when a false HTTP response is injected because, as mentioned above, the TCP layer will only accept the first segment that it receives (be it the false or the valid segment). When the forged response is shorter than and arrived before the valid response, the client then receives the byte stream that includes the forged response, followed by the tail of the valid response. The tail includes the data bytes having sequence numbers that immediately follow those of the forged response. By default, the response message body length is determined by the number of bytes received until the TCP connection is closed. This might be a problem for the injecting entity as the client will eventually receive a mixed HTTP response, which might yield unintended consequences. To avoid this problem, the injected response will usually include Content-Length or Transfer-Encoding headers that explicitly determine the end of the response. Thus, even if the TCP layer delivers the tail of the valid response to the HTTP layer, it will not be processed by the client.

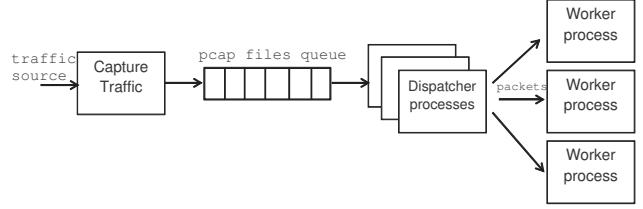


Figure 3: Depiction of the design of the monitoring system

3 Methodology

We now describe our methodology for collection and identification of TCP injection events.

3.1 Monitoring System

At the core of the collection of injection events was a monitoring system that eavesdropped on Internet traffic and identified these events. The monitoring system was deployed at the entry points of large networks (detailed in Section 4) and analyzed the bidirectional traffic that flowed in and out of those networks. The monitoring system was comprised of the following three stages (depicted in Figure 3). First, we captured the traffic using the 'netsniff-ng' tool [3] along with a Berkeley packet filter [25] to capture only HTTP traffic. The tool iteratively produced files comprising 200,000 packets each. These files were fed into a queue for processing by the next stage. To avoid explosion of the queue when the traffic rate exceeded the throughput of the next stages, the queue's length was bounded. Once the queue reached its limit, the capturing process was halted until the queue length decreased.

At the next stage each capture file was processed by a dispatcher process that read each packet in the file, removed the Ethernet header, and computed a hash on the IP addresses and TCP ports in such a way that packets of the same TCP session would have the same hash result. A packet's hash result was then used to choose one of several worker processes to handle that packet. In this way all packets of the same session were delivered to the same worker.

At the final stage each worker process grouped the packets it received into TCP sessions and stored each session in a data structure. For each received packet a worker checked all the packets of that session to determine whether the conditions for a packet race were met (the conditions are detailed in Section 3.2). If so, the last 30 packets of the session were written to a file, including their payload, for later analysis. See Section 3.3 for the ethics and privacy issues pertaining to the storage and analysis of packets.

The packet sessions were stored by each worker in a

data structure that is a least-recently-used cache with a fixed size. Once the cache reached the maximum number of sessions it can store, the session that was idle the longest was evicted from the cache. To simplify packet processing we did not use TCP signaling (SYN and FIN flags) to create a new session in the cache or evict an existing one. This design choice gave rise to the possibility that a session would be evicted even if still active. Nonetheless, as our experiments show, the caches were large enough so that the minimum idle time after which a session was evicted did not drop below 10 minutes — long enough to make the occurrences of active session evictions negligible. Note that even if such an eviction were to occur, packet races could still be detected in that session, since we treated the packets sent after the idle period as a new session and stored them in the cache. In this case, however, the packets of the session prior to the eviction would not be available for analysis.

3.2 Injection Detection

The detection logic of packet injection events is relatively straightforward. Our goal was to detect packet races within the session, namely, two packets that carry different payloads, but correspond to the same TCP sequence numbers. Usually these packets will arrive in quick succession. To make our code more efficient we checked for a race only between pairs of packets that were received within a time interval that does not exceed the parameter *MaxIntervalTime*. Throughout our data collection process we set *MaxIntervalTime* = 200msec. We believe that this value captures the vast majority of injection events as almost all round trip times on the Internet are below 400msec [18]. Indeed, nearly all of the time differences we observed between raced packets were below 100msec (see Section 5). Algorithm 1 in Appendix B details the procedure for race detection.

The procedure we used to identify packet races should, in theory, flag only events in which a third party injected rogue packets into the TCP session. However, interestingly, we observed numerous events which fulfill the above conditions but are not the result of a packet injection. We detail such occurrences in Appendix C.

3.3 Ethics and Privacy

As explained above, the monitoring system captures Internet user traffic. To minimize concerns about user privacy, the system stores only TCP sessions in which a packet race was detected. All other sessions are only cached briefly in the workers’ caches, after which they are permanently erased. Moreover, for each stored session, only the last 30 packets (at most) are saved. Earlier packets are dropped. This is in order to store only those

packets that are relevant to the analysis of the injection events while minimizing the chance that user privacy will be breached. Indeed, during our analysis no identifiable personal information was found in the stored sessions.

Throughout our research we were supervised by the networks’ administration teams, who reviewed and approved the code of the monitoring system and procedures for the analysis of the stored sessions. During the analysis the location and identity of users associated with IP addresses were never disclosed to us. Finally, we note that our monitoring system passively collected information; it never interfered or tampered in any way with the traffic.

3.4 Limitations

Our monitoring system cannot detect content alterations in which there is no race between the legitimate packet and the forged one. In particular, we cannot detect the following cases:

1. In-band changes in which the legitimate packet is changed in-place. In such cases the client only sees a forged packet.
2. Additions to the response in which an extra forged packet is sent such that it extends the HTTP response, but does not replace any legitimate part.
3. Drops of packets that are part of a valid HTTP response.

We monitored a large volume of traffic originating from diverse networks having tens of thousands of users (see Section 4). Nonetheless, as in any other study that involves uncontrolled traffic, our findings are only as diverse as the traffic we monitor. Namely, we cannot identify an injecting entity on the Internet if we do not monitor traffic that triggers an injection by that entity. Furthermore, the types of injections we have observed are dependent on the web traffic originating from the networks we monitored.

4 Data Sources

During our study we monitored the network traffic of four institutions. For each institution we monitored the Internet traffic (incoming and outgoing) of all its users. In all cases the same monitoring mechanism was used: traffic was copied to the monitoring system using a SPAN port out of a border switch. In all cases, we only monitored HTTP traffic, namely traffic having source port or destination port that equals 80.

Table 1 lists the characteristics of the monitored traffic sources. For each institution we list the number of users

Institution	User base	Monitoring period [week]	Traffic volume [Tb]	Number of sessions [Million]
University A	20,000	2	80	8
University B & University C	50,000	16	1400	120
Enterprise D	5,000	3	24	0.8

Table 1: Monitored traffic sources

who may use Internet connectivity in that institution. For a university this is the number of students and staff, and for an enterprise this is the number of employees. In addition, we list the length of time we monitored the traffic as well as the total volume traffic and number of sessions the monitoring system processed. In aggregate, we monitored the traffic of more than 75,000 users, while processing 1.4 petabits carried by 129 million HTTP sessions contacting servers having more than 1.5 million distinct IP addresses. The details of University B and C are displayed together since we monitored their traffic jointly on the same border switch. Enterprise D represents the main branch of a large hi-tech company. The monitored branch includes an extensive R&D division as well as the headquarter offices and the international marketing and sales divisions. All institutions wish to remain anonymous.

5 Injection Analysis

In this section we present an analysis of the injection events. In Section 5.1 we present an overview of the injections and highlight a few of them. Section 5.2 describes ways to automatically distinguish between the valid and forged packets. In Section 5.3 we explore the time differences between the raced packets. Section 5.4 characterizes the recurrence of injection events. Finally, Section 5.5 presents an investigation aimed at unveiling the entities behind the injection events.

5.1 Initial Investigation

In this section we refer to a TCP session into which a forged packet was injected as an *injected session*. We manually analyzed each injection event. We detected around 400 injection events that aim to alter web content². Although this is not a negligible number, it pales in comparison to the total volume of traffic we monitored to extract these events. This is attributed to the fact that most of the injected sessions were destined to web servers in the Far East, a region to which relatively

²We have also found hundreds of additional events that do not aim to alter web content; these events were related to caching and censorship.

little traffic is destined from the networks we monitored. Thus the relatively small number of injections. Nonetheless, these events were sufficient to gain substantial indications as to the different entities that practice forged content injection (Section 5.5).

We grouped the injection events into 14 groups based on the resource that was injected into the TCP session. In other words, two injections that forged the same content are placed in the same group. Representative (and anonymized) captures of the injected sessions can be found in [4]. For each injection group we publish up to 4 captures of injected sessions that are representative of their respective group. To preserve the anonymity of the users, in each capture we zeroed the client’s IP address as well as the IP and TCP checksum fields.

Table 2 lists the groups. For each group we list the following details:

1. Group name – an identifier that was given by us to that group. We selected the name either by the name of the site whose content was forged or by the name of a server the forged content directed us to.
2. Destination site(s) – the website(s) of the requested resource that was forged. There may be several such sites for a single group.
3. Site type – the category of the destination site(s)
4. Location – the country of the IP address of the destination server³
5. Injected resource – the type of forged content that was injected
6. Purpose – the aim of the injection

It is evident from Table 2 that the majority of injected sessions we observed were to web servers located in China. We note that the networks we monitored are *not* located in China or the Far East, but in a Western country. The proportion of HTTP traffic destined to China in the monitored networks is only about 2%. This is a first indication that the majority of entities that injected the forged

³Note that this country might be different than the nationality of the entity that owns the destination site.

Group name	Destination site(s)	Site type	Location	Injected resource	Purpose
szzhengan	wa.kuwo.cn	Ad network	China	A JavaScript that appends content to the original site	Malware
taobao	is.alicdn.com	Ad network	China	A JavaScript that generates a pop-up frame	Advertisement
netsweeper	skyscnr.com	Travel search engine	India	A 302 (Moved) HTTP response	Content filtering
uyan	uyan.cc	Social network	China	A redirection using 'meta-refresh' tag	Advertisement
icourses	icourses.cn	Online courses portal	China	A redirection using 'meta-refresh' tag	Advertisement
uvclick	cnzz.com	Web users' statistics	Malaysia/China	A JavaScript that identifies the client's device	Advertisement
adcpc	cnzz.com	Web users' statistics	Malaysia/China	A 302 redirection to a JavaScript that opens a new window	Advertisement
jiathis	jiathis.com	Social network	China	A redirection using 'meta-refresh' tag	Advertisement
server erased	changsha.cn	Travel	China	Same as legitimate response but the value of HTTP header 'Server' is changed	Content filtering
gpwa	gpwa.org	Gambling	United States	A JavaScript that redirects to a resource at qpwa.org	Malware
tupian	www.feiniu.com www.j1.com	e-commerce	China	A JavaScript that directs to a resource at www.tupian6688.com	Malware
mi-img	mi-img.com	Unknown	China	A 302 redirection to a different IP	Malware
duba	unknown	Unknown	China	A JavaScript that prompts the user to download an executable	Malware
hao	02995.com	Adware-related	China	A 302 (Moved) HTTP response	Advertisement

Table 2: Injection groups and their characteristics

content we observed reside in China (we investigate the injectors’ identity in Section 5.5).

Seven injection groups are aimed at injecting advertisements to web pages. An analysis of the injected resources shows similarities between the various groups. These similarities might indicate that the injections are done by the same entity or at least by different entities that use the same injection mechanism or product. The injection groups ‘icourses’, ‘uyan’, and ‘jiathis’ all used the HTML meta refresh tag to redirect the user to a different URL. In all cases, the redirection was to Baidu (a Chinese search engine) using the URL `www.baidu.com/?tn=95112007_hao_pg`. The URL includes a referral tag that identifies `hao123.com` – a well-known adware-related site – as the referring site. The referral tag is possibly used by Baidu to pay `hao123` for referring traffic to it. In one case, the redirected URL included a search keyword for a clothing chain store. Interestingly, another injection group, ‘hao’, referred the user to `hao123.com` itself, but using a different mechanism – an HTTP 302 response.

Surprisingly, five injection groups showed strong indications that the aim of the injector was malicious. One such group is ‘gpwa’. The injections in this group target the traffic to `gpwa.org`. The forged content here includes a JavaScript that refers to a resource having the same name as the one originally requested by the user, but the forged resource is located at `qpwa.org`, a domain that is suspiciously similar to the legitimate domain. The forged domain is registered to a Romanian citizen, who appears to be unrelated to the organization that registered the domain `gpwa.org`. These are strong indications of malicious intent. As of May 2016 the web server of `qpwa.org` is still active at a web hosting provider based in the US, however we have not been able to retrieve from it the malicious script.

The injections in the ‘duba’ group add to the original content of a website a colorful button that prompts the user to download an executable from a URL at the domain `duba.net`. The executable is flagged as malicious by several anti-virus vendors.

Another malicious injection group is ‘mi-img’. In these injected sessions the client, which appears to be an Android device, tries to download an application. The injected response is a 302 redirection to another IP address (no domain name is specified). According to BotScout [2] – an online bot database – this forged IP address is known to be a bot. We retrieved the application from this IP address. The downloaded apk file is flagged by Fortinet’s antivirus as a malware called ‘Android/Gepew.A!tr’.

Another injection group worth mentioning is ‘server erased’. In this group injections were identical to the legitimate response but instead of original value of

the Server HTTP header, e.g., `nginx/1.2.7`, the string ‘*****’ appeared. This is as if to prevent identification of the web server’s software. We assume that this injection is due to a security measure at the network operator. The HTTP specification [14] indeed recommends that Server header be configurable.

5.2 Distinguishing the Forged Response from the Valid One

Identifying a race between two packets is a relatively straightforward task. However, without a priori knowledge of the legitimate content expected from the server, automatically distinguishing the forged packet from the legitimate one is not trivial. Nonetheless, in the following we list a few rules that worked well for this difficult task.

IP identification In many operating systems, such as Windows and Linux [16], the IP identification value equals a counter that is incremented sequentially with each sent packet. In some operating systems there is a single global counter for all sessions. In others, there is a separate counter for each destination. Indeed, our observations show that in most injected sessions the IP identification values of the packets sent by the web server are either monotonically increasing (when the counter is global) or consecutively increasing (when there is a counter per destination). In most of the injection events we observed that the injecting entity made no attempt to make the identification value of the forged packet similar to the identification values of the other packets sent by the server. In Appendix D we detail a few of the (failed) attempts of the injecting entity to mimic the Identification field of the legitimate packet it aims to displace.

We formulate the following rule to determine which of the two raced packets is the forged one: the forged packet is the one that has the largest absolute difference between its identification value and the average of the identification values of all the other packets (except the raced one).

For all injection events, we manually identified the forged packet according to its content and compared it to the corresponding identification that used the above rule. The comparison reveals that the rule is accurate about 90% of the time. This is a fairly accurate measure considering that it is not based on the payload of the raced packets.

IP TTL The IP TTL value in a received packet is dependent on the initial value set by the sender and the number of hops the packet has traversed so far. Thus, it is unusual for packets of the same session to arrive at the

client with different TTL values. Therefore, if the raced packets have different TTL values we can use them to distinguish between the two packets. From our observations, the injecting entity often made no attempt to make the TTL value of the forged packet similar to the TTL values of the other packets sent by the server. Similarly to the case of the IP identification rule above, we identify the forged packet using the following rule: the forged packet is the one that has the largest absolute difference between its TTL value and the average of TTL values of all the other packets (except the raced one).

Manual analysis of the injection events reveals that the TTL rule correctly identified the forged packet in 87% of all injection events. The TTL rule concurs with the IP identification rule above in 84% of all injection events. We thus conclude that the TTL and identification values can serve to effectively distinguish the forged packet from the valid packet.

We note that our finding that the TTL and Identification fields of the forged packets have abnormal values generally agrees with findings on censorship-related injections which also show that censoring entities do not align the TTL and Identification values with those of the legitimate packets (e.g., [8]).

5.3 Timing Analysis

The race between the forged and legitimate packets can also be characterized by the difference in their arrival times. By arrival time we mean the time at which the packet was captured by the monitoring system. Since the system captures traffic at the entrance to the edge network close to the client, it is reasonable to assume that these times are very close to the actual arrival times at the end client. For each injection event we calculate the difference between the arrival time of the legitimate packet and the arrival time of the forged packet. A negative difference means that the forged packet “won” the race, and a positive difference means that the legitimate packet “won”. The histogram of the time differences of all the injection events we observed are shown in Figure 4.

It is evident from Figure 4 that in most injection events the forged packet wins the race. In only 32% of the events does the legitimate packet arrive first. This result strengthens our initial assumption that the decision to inject a forged packet is made according to the HTTP request sent by the client. This means that the injecting entity can send the forged packet well before the server sends the legitimate packet, as the client’s request still needs to travel to the server. Still, even in such a case, in a non-negligible portion of events, the forged packet loses the race. This may indicate injections that occurred very close to the server. Alternatively, it may indicate that

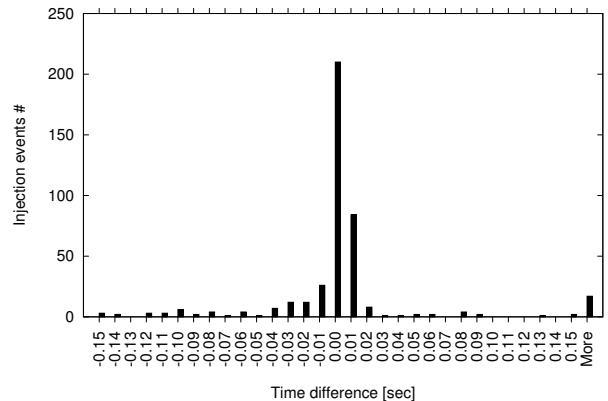


Figure 4: Arrival time difference between the forged and legitimate packets

in some cases the decision to inject the packet is made at the time the response from the server is encountered. In the latter case, the forged packet is at a distinct disadvantage as it starts the race lagging behind the legitimate packet. In many cases in which the forged packet won the race, the legitimate packet arrived very soon after, in less than 10msec.

5.4 Repeatability

All injection groups were observed for only a short period of time, usually one to three days, after which they were not detected again by our monitoring system. A few injection types were even encountered only once. No long-term (3 days or more) injections were observed by our monitoring system⁴.

We next tried to reproduce the injection events we observed. This attempt was made several weeks after the initial observations of the injections. For each injection event we extracted the HTTP request that triggered the injection. We then sent from the edge network in which the injection originally occurred the same HTTP request (following a proper TCP 3-way handshake) to the destination web server. We sent each request 1000 times. This is with the aim to reproduce the injections even if they do not occur for every request. We captured the resulting TCP sessions and searched for injections. We were not able to reproduce any of the injection groups.

Following the initial publication of this work an effort independent of our own to reproduce the injections had more success [17]. The ‘gpwa’ and ‘hao’ injections were successfully reproduced. However, the author of [17] has

⁴The only long-term injections we did observe were related to censorship and caching. These injections were the only ones we were able to reproduce.

not been able to reproduce those injections again in a second attempt made a few weeks later. Moreover, when the injections were observed by [17] they were not always reliable. For one of the resolved IP addresses (for the destination site’s domain name) the injections were observed only 30% of the time (this information was given to us via personal communication by the author of [17]).

From the above findings we surmise that, in general, injections by on-path entities may be intermittent; namely, the injecting entity injects forged content to a particular site for only a short period of time before moving on to other sites. Moreover, when an injector is active for a web site it may target only a portion of the HTTP requests. This might be motivated by the desire of the injector to stay “under the radar”. It is plausible that injecting forged content to a site for only a short period of time might go unnoticed by the users and site owners, or at least would not cause them to expend effort investigating the forged content’s origin.

The injections we found were triggered by an HTTP request to specific resources which in most cases were not the main page of the site. This leads us to assume that an effort to actively seek other sites for possible injections may be computationally too expensive as we would need to crawl those entire sites.

5.5 Who is Behind the Injections?

We finally turn our attention to the culprits behind these injection events. In general, it is difficult to unveil these entities as there is no identifying information in the injected content. Nonetheless, we can get indications as to the identity of the injecting entities by trying to detect the autonomous system from which the forged packet originated. We assume that the entity that operates this autonomous system is the entity responsible for the injection.

Note that the analysis thus far shows strong indications that the injections do not originate at the web servers themselves. First, the injected responses had anomalous IP ID and TTL values. To bring this about an injecting rogue software on the end server would need to circumvent the standard TCP/IP stack as it sends packets. While this is possible it would require the injecting software elevated privileges and more complex logic to send the injected responses. Such elevated privileges would have also allowed the injector to block the valid response and eliminate the possibility of a race altogether. Second, most of the injected packets “win” the race. An attacker injecting packets from the end server does not have a distinct advantage to win the race. Therefore it is reasonable to assume that in such a case the race would have been more even. Third, to the best of our knowledge there is no malware that injects packets out-of-band. All known

malware that aim to alter traffic on the machine they reside alter the the actual packets to be sent (usually by simply injecting code to the sending process or hooking the suitable system services).

We note that we ruled out the possibility that the edge network operators serving the networks we monitored are responsible for the injections. We verified this by speaking directly with the network operators’ administrators and sharing with them the injections we found.

Since the injections were not reproducible during this analysis, we cannot employ the oft-used traceroute-like procedure to locate the injector [22, 8, 24]. In this procedure the packet triggering the injection is repeatedly sent with increasing TTL values until the forged response is triggered, thereby revealing the location of the injector. To identify the injecting entities we resort to the following procedure:

1. Estimate the number of hops the forged packet traversed: this estimation relies on the packet’s TTL value. Specifically, it relies on there being a significant difference between the default initial TTL values set by the major operating systems [29]: in general, the differences between those initial values are larger than the length of most routes on the Internet. The default initial TTL values of the major operating systems are 32, 64, 128 and 255. This means, for example, that if a packet is received with a TTL value of 57, the initial TTL value of that packet was likely to be 64 and the number of hops traversed was likely to be 7. If the estimated number of hops is larger than 30 or smaller than 3⁵, we assume the estimation is incorrect and stop the analysis.
2. Identify the path from the destination server to the client: the actual path from the server to the client cannot be known without an agent in the server’s network. Instead, we use the path from the client to the server while assuming that the routing on this path is symmetric. We identify the path from the client to the server by using a ‘traceroute’ tool. The traceroute used a TCP syn packet with destination port 80. We found that such a packet triggers responses from most routers and servers.
3. Infer the hop along the above path from which the forged packet was injected: using the estimated number of hops the forged packet traversed and the estimated path it traversed, we can now infer the hop on the path from which the packet was sent.

⁵Nearly all routes on the Internet are shorter than 30 hops [21]. Additionally, it is very unlikely that the injecting third party resides less than 3 hops away since the first couple of hops reside within the edge networks we were monitoring.

Injection group	Web server's AS number	Suspected injecting AS number
xunlei	17816	17816
szzhengan	4134	4134
taobao	4837	4837
uvclick	38182	38182
adcpc	38182	38182
server erased	4134	4134
GPWA	6943	6943
tupian	4812	4812

Table 3: The autonomous system numbers in which the injected web servers reside and in which the suspected injecting entities reside

4. Identify the autonomous system the injecting hop belongs to: given the IP address of the hop, we can now identify the autonomous system to which it belongs in order to reveal the entity responsible for injecting the packet. To this end we leveraged public databases that hold current BGP advertisements: this allows us to identify the autonomous system that advertises the given IP address. BGP advertisements for mapping of IP addresses to autonomous systems are known to be more precise and up-to-date than Internet route registries [23].

It should be noted that this procedure has the following caveats:

1. The initial TTL value of the injected packet may not be one of the common default values. In such cases, this analysis can not be carried out. In particular, based on the TTL values of the injected packet, we conclude that this is indeed the case for the injections in the groups 'jiathis', 'uyan', 'mi-img', and 'icourses'.
2. Not all routes on the Internet are symmetric. If the path from the client to the server is not symmetric, the analysis will produce an incorrect result. We address this issue in the next subsection.
3. The implicit assumption of this procedure is that the injecting machine resides on-path. Strictly speaking, this need not be the case. An on-path machine monitoring the traffic can trigger the injection from a remote machine. In such a case the forged packet will travel on an entirely different path than the legitimate packets.

In Table 3 we list the results of the above analysis. For each injection group, we list the autonomous systems

AS number	Operator
17816, 4837	China Unicom
4134, 4812	China Telecom
38182	Extreme Broadband (Malaysia)
6943	Information Technology Systems (US)

Table 4: The operators for each suspected injecting autonomous system

in which the destination sites reside and the autonomous systems suspected of the injections. The table lists only injection groups for which the analysis can be performed; namely, the estimated number of hops the injected packet traversed is not larger than 30 and not smaller than 3, and it is also not larger than the path between the client and server.

In all cases where the above analysis succeeded, it indicated that the forged content was injected 2-5 hops away from destination site. Since the injection groups are largely independent we believe that this is a signal that the assumptions we made throughput the above analysis are not far off. In all cases the injector is located in the very same autonomous system where the destination site resides. Indeed, this is the most reasonable location for an injector to be in order to alter content for all web users accessing the targeted site.

Table 4 lists for each suspected injecting autonomous system the organization that operates it. It is worth noting that two of the largest network operators in China – China Unicom and China Telecom – are suspected of practicing content injections. Moreover, the autonomous systems of these operators originate injections of different groups. This might imply that more than one injector mechanism is deployed in these autonomous systems.

The operator of the suspected autonomous system for the 'gpwa' group is Information Technology Systems. In this particular case, this is the organization that is responsible for the content of the destination site for these injections – gpwa.org. Since there are strong indications that the injections of this group are malicious (see discussion in Section 5.1), we assume that the attacker compromised a router in the suspected autonomous system.

Using a traceroute from the server-side

As noted above, a caveat of the above analysis is that we used traceroutes from the client to the server while assuming this route is symmetric. This is a necessity since we cannot execute a traceroute to the client from the actual server. To address this caveat we leveraged RIPE Atlas [26]. This is a global network comprised of thousands of probes hosted throughout the Internet. Each probe can

AS number	Injection groups
4812	tupian
4134	szzhengan, server erased
4808	uyan, icourses, jiathis

Table 5: Autonomous systems which host RIPE Atlas probes and the corresponding injection groups that forged traffic of web servers residing in those autonomous systems

be instructed to execute a measurement out of a predetermined set that includes ping, DNS query, HTTP request and traceroute. RIPE Atlas hosts 6 probes in 3 autonomous systems that host destination sites the content of which was forged. The autonomous systems and the corresponding injection groups that forged content for destination sites residing in each autonomous system are listed in Table 5.

For each of the 6 probes we executed a traceroute from it to the edge network where the corresponding injection events were identified. We then employed the procedure we described above on these new traceroutes. We note that using these traceroutes may still not be without error. The probes indeed reside in the autonomous systems that host the destination site; however, we cannot guarantee that their route to the client is the same as the route from the destination site. Specifically, the traffic from the probe may exit the autonomous system through a different point than the traffic originated from the site.

The traceroute from each of the 3 autonomous systems to the corresponding edge network were *different* than the opposite routes from the edge networks to those autonomous systems. Nonetheless, in all cases, a pair of routes in opposite directions traversed the same autonomous systems with the exception of one Tier-1 autonomous system; namely, each route traversed a different Tier 1 operator (for example, the route between the client and the server traversed Level 3’s AS while the route in the opposite direction traversed Cogentco’s AS). The other autonomous systems on the routes were the same; this is why the outcome of the analysis with these routes was the same as for the routes in the opposite direction. The analysis for the ‘szzhengan’ and ‘server erased’ injections yielded the same suspected autonomous system – 4134, while the analysis for the ‘tupian’ injections yielded a different autonomous system – 4134 instead of 4812 found by the previous analysis. Nonetheless, these autonomous systems are siblings operated by the same company – China Telecom.

The injecting groups that correspond to destination sites residing in autonomous system 4808 – ‘uyan’, ‘icourses’, and ‘jiathis’ – were set with an unknown ini-

tial TTL value (namely the estimated number of hops was larger than 30 or smaller than 3); hence the analysis cannot be performed on them.

6 Proposed Mitigation

The best mitigation against TCP injection attacks is simply to use HTTPS. Unfortunately, this is not always subject to the discretion of the user. Many web sites still do not support HTTPS [5]. A user wishing to access a website that does not support HTTPS must resort to the unprotected HTTP. Moreover, about 17% of the Alexa Top 500 websites still serve a login page over HTTP but submit the users password over HTTPS [30]. This setup allows an on-path entity to steal a user’s login credentials by injecting a false login page. In this section we present a client-side mitigation measure that monitors the incoming HTTP traffic and blocks injected forged TCP segments, thereby defending the user even if he must use HTTP.

A naive mitigation measure is to simply apply the procedure described in Algorithm 1 on the monitored traffic in order to identify packet races. Nonetheless, such an approach means that every incoming packet must be delayed for 200msec. Such a delay is necessary in order to make sure a given packet is not an injected packet forging a legitimate one. Only after 200msec have passed with no race detected can we accept the packet. Such an approach incurs noticeable delay on the incoming traffic and degrades the user’s browsing experience. This approach, however, by definition, ensures that all injected packets will be identified and blocked. In Section 6.1 we detail our experimental results with such an approach. We use these results as a benchmark for the next mitigation approach.

An improved approach is to take advantage of the insights we presented in Section 5.2, where we showed that for the vast majority of the injected packets, the values of the TTL and Identification fields in the IP header do not correspond to the respective values of the legitimate packets of the session. This insight can be leveraged to improve the naive mitigation measure such that only packets with abnormal TTL or Identification values will be delayed for 200msec, and only for those packets will we try to detect a race. This way only suspicious packets are delayed.

Algorithm 2 in Appendix E details the improved mitigation algorithm. Note that this algorithm will be effective only if the forged packets exhibit anomalous TTL or Identification values as compared to the legitimate packets in the injected session. We note that it is possible for an injector to inject a packet with values that will not appear anomalous, as in most likelihood it can also inspect the traffic sent by the web server. Anomalous TTL

and identification values have also been observed in the censorship-related state-sponsored injections [8]. This indicates that aligning the TTL and identification values to the legitimate values might not be trivial to implement. Indeed, aligning the identification value requires that injector keep track of the identification values of packets sent by the web server for every potential session that may be injected, well before the actual injection decision is made. This may require a substantial addition of memory space and computational overhead. If the injector does align the TTL and identification values, the improved mitigation algorithm we propose will not be effective and the naive approach must be used.

6.1 Experiments

We now detail our experiments to evaluate the two mitigation algorithms – the naive and improved algorithms. We evaluate the algorithms using two measures:

1. Web page load time increase – this measure shows the increase of time it takes to load a web page as compared to the case where no mitigation measure is employed. This measures the extent to which the algorithm degrades the user’s experience.
2. False negatives – this measure counts how many injections are not identified. This measures the effectiveness of the algorithm.

We evaluated the algorithms against two data sets:

1. Benign data set – this data set includes traffic of benign web browsing having no content injection. We used the 200 most popular sites from Alexa’s list [1]. From these sites we used the ones for which majority of their objects are fetched using HTTP (rather than HTTPS). There are 136 of these sites that met this criterion.
2. Injected data set – This data set includes the injected sessions we captured throughout our observations.

The two algorithms were evaluated on the benign data-set to measure the web page load time increase. We browsed each website using PhantomJS. We inspected the incoming traffic while leveraging the NFQUEUE target of Linux iptables [6]. We measured the load time of each website 5 times and recorded the smallest load time value to disregard intermittent network delays. We compared these load times to the load times where no mitigation algorithm is deployed.

The two algorithms were evaluated on the injected data set to measure the false negative events, i.e., the injections that were missed. Table 6 summarizes the findings. It is evident that the naive algorithm imposes

Algorithm	Load time increase	False Negative
naive	120%	0%
improved	12%	0.3%

Table 6: The performance of the two mitigation algorithms.

a considerable increase in page load time – 120%. In contrast, the improved algorithm incurs a mere 12% increase, while having a negligible false negative rate of 0.3%.

7 Related Work

The practice of Internet traffic alteration has been studied in several works [20, 34, 28, 39], all of which have employed the server-centric approach described in the Introduction.

In [20, 34] the authors deployed a website that directs users to about 20 back-end servers that deliver a Java applet. The applet runs a series of tests which try to fetch predetermined content. The analysis found many web proxies of several categories, the most popular of which are anti-virus software installed on the end clients, HTTP caches and transcoders deployed by ISPs, and security and censor proxies deployed by enterprises and countries. Ref. [34] identifies two ISPs that employ HTTP error monetization, and one that injects advertisements into all HTTP connections.

In [28] the authors set up a web server that delivers the same content from a handful of different domains. The content includes a JavaScript code that runs when the page is loaded in the client’s browser and reports any detected changes to the web page. It found that most changes to the content were made indiscriminately regardless of the originating domains. Most of the content modifications were due to software installed locally on the end clients or due to security gateways deployed at enterprises. Other modifications were due to ISPs that compressed content delivered to their users. Additionally, 4 ISPs and a company that provides free wireless service were identified as injecting advertisements to web pages their customers visit.

In [39] the authors leveraged the online advertising infrastructure of several ad networks to spread a specially crafted Flash-based advertisement that runs a JavaScript code and retrieves a preconfigured measurement page while reporting back any change made to it. Almost 1000 page alteration events were detected; however, the portion of events for which ISPs are responsible is unknown.

The authors of [38] investigate inflight modifications of traffic from an unnamed popular Internet search ser-

vice. In contrast to the abovementioned works, here the changes were detected by the IP address the client contacted, which was different than the addresses owned by the search service. This work found 9 ISPs that proxy their customers' traffic destined to the search service. The redirection to the proxy is done by resolving the DNS name of the service to the IP address of the proxy.

A considerable body of work deals with censoring countries and the mechanisms they use to censor Internet traffic. The authors of [33] have categorized the mechanisms of the censorship employed by different countries. It is noted that China and Thailand use out-of-band devices to send forged packets, which are usually HTTP 302 redirection, or a TCP reset.

In [35] it was shown that several ISPs enforce usage restrictions of their networks by actively terminating undesirable TCP connections. The authors note that this is done by sending forged TCP resets out-of-band. They then leverage this insight – much as we do in the current work – to identify these forged resets. Nonetheless, the detection conditions are different than the ones we used since the forged TCP reset has no payload to spoof; hence, the detection conditions mainly revolve around the arrival time and sequence number of the reset segment as compared to those of other segments in the connection.

The authors of [13] discuss attacks that employ out-of-band injection of forged DNS responses. To mitigate the effects of such attacks it is suggested that the resolver wait after receiving an initial reply to allow a subsequent legitimate reply to also arrive. In particular, the resolver should wait for another reply if the first reply arrived sooner than half of the expected RTT since the query was issued or if the TTL field in the IP header does not have the expected value. If indeed two replies eventually arrive, this indicates an attack.

8 Conclusions

In this work we reveal a new side to the practice of false content injection on the Internet. Previously, discussion on this practice focused on edge ISPs that limit their misdeeds to the traffic of their customers. However, we discovered that some network operators inject false content to the traffic of predetermined websites, regardless of the users that visit them. Our work leverages the observation that rogue content injection is done out-of-band. It can hence be identified while monitoring an edge network in which the victim clients reside. Our analysis is based on extensive monitoring of a large amount of Internet traffic. We reveal 14 groups of content injections that primarily aim to impose advertisements or even maliciously compromise the client. Most of the financially-motivated false content injection we observed originated

from China. Our analysis found indications that numerous injections originated from networks operated by China Telecom and China Unicom – two of the largest network operators in Asia.

Acknowledgments

We would like to thank Hank Nussbacher and Eli Beker, whose cooperation made this research possible. We also thank Erik Hjelmvik for his efforts to independently reproduce the injections.

References

- [1] Alexa. <http://www.alexa.com/>.
- [2] BotScout. <http://botscout.com/>.
- [3] netsniff-ng toolkit. <http://netsniff-ng.org>.
- [4] Representative captures of the injected sessions. <http://www.cs.technion.ac.il/~gnakibly/TCPInjections/samples.zip>.
- [5] SSL/TLS analysis of the Internet's top 1,000,000 websites. https://jve.linuxwall.info/blog/index.php?post/TLS_Survey.
- [6] Using NFQUEUE and libnetfilter.queue. https://home.regit.org/netfilter-en/using-nfqueue-and-libnetfilter_queue/.
- [7] ANDERSON, N. How a banner ad for H&R Block appeared on apple.com. <http://arstechnica.com/tech-policy/2013/04/how-a-banner-ad-for-hs-ok/>.
- [8] ANONYMOUS. Towards a comprehensive picture of the great firewalls dns censorship anonymous. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)* (2014).
- [9] ARYAN, S., ARYAN, H., AND HALDERMAN, J. A. Internet censorship in Iran: A first look. In *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet* (2013).
- [10] BODE, K. Mediacom Injecting Their Ads Into Other Websites. <http://www.dsreports.com/shownews/112918>.
- [11] CLAYTON, R., MURDOCH, S. J., AND WATSON, R. N. Ignoring the great firewall of China. In *Privacy Enhancing Technologies* (2006), Springer, pp. 20–35.
- [12] DIERKS, T., AND RESCORLA, E. The transport layer security (TLS) protocol version 1.2. RFC 5246, August 2008.
- [13] DUAN, H., WEAVER, N., ZHAO, Z., HU, M., LIANG, J., JIANG, J., LI, K., AND PAXSON, V. Hold-on: Protecting against on-path dns poisoning. In *Proc. Workshop on Securing and Trusting Internet Names, SATIN* (2012).
- [14] FIELDING, R., AND ET AL. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999.
- [15] FIELDING, R., AND RESCHKE, J. Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. RFC 7230, June 2014.
- [16] HERZBERG, A., AND SHULMAN, H. Security of patched DNS. In *Computer Security—ESORICS 2012*. Springer, 2012, pp. 271–288.
- [17] HJELMVIK, E. Packet injection attacks in the wild. <https://www.netresec.com/?page=Blog&month=2016-03&post=Packet-Injection-Attacks-in-the-Wild>.

- [18] HUFFAKER, B., PLUMMER, D., MOORE, D., AND CLAFFY, K. Topology discovery by active probing. In *Symposium on Applications and the Internet (SAINT)* (Jan 2002), pp. 90–96.
- [19] KEARNEY, R. Comcast caught hijacking web traffic. <http://blog.ryankearney.com/2013/01/comcast-caught-intercepting-and-altering-your-web-traffic/>.
- [20] KREIBICH, C., WEAVER, N., NECHAEV, B., AND PAXSON, V. Netalyzr: illuminating the edge network. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (2010), pp. 246–259.
- [21] LEGUAY, J., LATAPY, M., FRIEDMAN, T., AND SALAMATIAN, K. Describing and simulating Internet routes. In *NETWORKING 2005*. Springer, 2005, pp. 659–670.
- [22] LEVIS, P. The collateral damage of internet censorship by DNS injection. *ACM SIGCOMM CCR* 42, 3 (2012).
- [23] MAO, Z. M., REXFORD, J., WANG, J., AND KATZ, R. H. Towards an accurate AS-level traceroute tool. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2003), pp. 365–378.
- [24] MARCZAK, B., WEAVER, N., DALEK, J., ENSAFI, R., FIELD, D., MCKUNE, S., REY, A., SCOTT-RAILTON, J., DEIBERT, R., AND PAXSON, V. An analysis of China’s “Great Cannon”. In *5th USENIX Workshop on Free and Open Communications on the Internet (FOCI 15)* (2015).
- [25] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter USENIX Conference* (1993), USENIX Association.
- [26] NCC, R. RIPE Atlas. <https://atlas.ripe.net>.
- [27] POSTEL, J. Transmission control protocol. RFC 793, September 1981.
- [28] REIS, C., GRIBBLE, S. D., KOHNO, T., AND WEAVER, N. C. Detecting in-flight page changes with web tripwires. In *NSDI* (2008), vol. 8, pp. 31–44.
- [29] SIBY, S. Default TTL (Time To Live) Values of Different OS. <https://subinsb.com/default-device-ttl-values>, 2014.
- [30] SILVER, D., JANA, S., BONEH, D., CHEN, E., AND JACKSON, C. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 449–464.
- [31] TOPOLSKI, R. NebuAd and partner ISPs: Wiretapping, forgery and browser hijacking, June 2008. http://www.freepress.net/files/NebuAd_Report.pdf.
- [32] TOUCH, J., MANKIN, A., AND BONICA, R. The TCP authentication option. RFC 5925, June 2010.
- [33] VERKAMP, J.-P., AND GUPTA, M. Inferring mechanics of web censorship around the world. *Free and Open Communications on the Internet, Bellevue, WA, USA* (2012).
- [34] WEAVER, N., KREIBICH, C., DAM, M., AND PAXSON, V. Here be web proxies. In *Passive and Active Measurement* (2014), Springer, pp. 183–192.
- [35] WEAVER, N., SOMMER, R., AND PAXSON, V. Detecting forged TCP reset packets. In *NDSS* (2009).
- [36] WEINSTEIN, L. Google Hijacked – Major ISP to Intercept and Modify Web Pages. <http://lauren.vortex.com/archive/000337.html>.
- [37] XU, X., MAO, Z. M., AND HALDERMAN, J. A. Internet censorship in China: Where does the filtering occur? In *Passive and Active Measurement* (2011), Springer, pp. 133–142.
- [38] ZHANG, C., HUANG, C., ROSS, K. W., MALTZ, D. A., AND LI, J. Inflight modifications of content: Who are the culprits. In *Workshop of Large-Scale Exploits and Emerging Threats (LEET11)* (2011).
- [39] ZIMMERMAN, P. T. Measuring privacy, security, and censorship through the utilization of online advertising exchanges. Tech. rep., Princeton University, June 2015.

A “Ack storm” due to TCP Injection

An “Ack storm” occurs when the injected segment causes the receiver to send an acknowledgment for data bytes having sequence numbers that were not yet sent by the peer. This acknowledgment is dropped by the peer, triggering it to respond by resending an earlier Ack, which may in turn trigger a retransmission by the receiver. The retransmitted segment will include again an acknowledgment for the yet to be sent sequence numbers and so forth. Such a “ping-pong” exchange, if run long enough, will cause the connection to timeout and reset. In many cases this is undesirable for the injector as it will interfere with the flow of traffic on the connection. An “Ack storm” can subside if the peer eventually sends data bytes having sequence numbers that correspond to those of the forged data bytes injected by the third party.

B Injection Detection Algorithm

Algorithm 1 details the procedure for detecting packet races. This algorithm is executed by each worker process upon the receipt of a new packet. In the following, CP denotes the currently received packet and S denotes the set of packets received so far as part of the session of CP . $P(f)$ denotes the value of parameter f of packet P . If parameter f is a field of TCP or IP, it is denoted by the protocol and field names, e.g., $P(IP_total_length)$ denotes the value of the field Total Length in the IP header of packet P . The algorithm returns True if and only if a race is detected.

In Algorithm 1, line 1 iterates over the previously received packets of the current session. Line 2 verifies that the two considered packets have been received within a time interval that does not exceed the parameter $MaxIntervalTime$. Lines 5 and 6 compute the total lengths of the TCP and IP headers of each of the two packets. Lines 7 and 8 compute the payload size of each of the two packets. Lines 9 and 10 compute the TCP sequence number of the last byte delivered in the payload in each of the two packets. Lines 11 and 12 check for a sequence number overlap between the two packets. Line 15 checks whether the overlapped payload is different. If it is, a race is detected and the algorithm returns True.

To avoid false positives, we did not consider the following packets (not shown in Algorithm 1):

1. Checksum errors – packets that have checksum errors either in the TCP or IP headers will clearly have

```

Input: CP, S
1 foreach  $OP$  in  $S$  do
2   if  $CP(t) - OP(t) > MaxIntervalTime$  then
3     | continue;
4   end
5    $CP(headers\_size) = CP(IP\_header\_length) + CP(TCP\_data\_offset)*4;$ 
6    $OP(headers\_size) = OP(IP\_header\_length) + OP(TCP\_data\_offset)*4;$ 
7    $CP(payload\_size) = CP(IP\_total\_length) - CP(headers\_size);$ 
8    $OP(payload\_size) = OP(IP\_total\_length) - OP(headers\_size);$ 
9    $CP(top\_sequence\_number) = CP(TCP\_sequence\_number) + CP(payload\_size);$ 
10   $OP(top\_sequence\_number) = OP(TCP\_sequence\_number) + OP(payload\_size);$ 
11  if  $CP(top\_sequence\_number) > OP(TCP\_sequence\_number)$  then
12    if  $OP(top\_sequence\_number) > CP(TCP\_sequence\_number)$  then
13       $bottom\_overlap = MAX(CP(TCP\_sequence\_number), OP(TCP\_sequence\_number));$ 
14       $top\_overlap = MIN(CP(top\_sequence\_number), OP(top\_sequence\_number));$ 
15      if  $CP(TCP\_payload)[bottom\_overlap:top\_overlap] \neq$ 
16         $OP(TCP\_payload)[bottom\_overlap:top\_overlap]$  then
17          | return True;
18        end
19      end
20    end
21  end
22 return False;

```

Algorithm 1: Race detection algorithm

a different payload than that of their retransmission.

2. TCP reset – reset packets can carry data payloads for diagnostic messages which are not part of the regular session’s byte stream.

C False Positives

There were numerous events in which the race identification algorithm (described in Appendix B) of our monitoring system identified a race that was not due to a forged packet injection. In the following we describe these events and why they occur:

Retransmissions with different content As per the TCP specification [27], the payload of retransmitted segments must have the same content as the payload of the original segment. In practice, however, this is not always the case, and retransmitted segments sometimes carry slightly different content, for the following reasons:

- Load balancing – some websites serve HTTP requests using more than one server. Usually, a front-end load balancer redirects the HTTP requests according to the current load on each web server. It is sometimes desirable that the same server serve all HTTP requests coming from the same client. To facilitate this, the first HTTP response sent to

a client sets a cookie containing the identity of the server chosen to serve the client from now on. Subsequent requests from that client will include this server ID and allow the load balancer to redirect those requests to that server. If the first HTTP response needs to be retransmitted, some load balancers might, at the time of the retransmission, choose a different web server than the one they originally chose when the response was first transmitted. This results in a different cookie value set in the retransmitted response. Examples of websites that exhibit such behavior are wiley.com and rottentomatoes.com.

- Accept-Ranges HTTP header – the HTTP 1.1 specification [15] allows a client to request a portion of a resource by using the Range header in the HTTP request. It may do so in cases where the web server has indicated in previous responses its support of such range requests. Such support is indicated by the Accept-Ranges header. We observed cases where a web server sent an HTTP response which included 'Accept-Ranges: none', indicating that the server is unwilling to accept range requests, while in a retransmission of the same response the header was replaced by 'Accept-Ranges: bytes', indicating that it is willing to accept range requests having units of bytes. This happened when the retrieved

resource spanned multiple TCP segments. Presumably, the intention of the server is to allow the client to retrieve a portion of a resource when network loss is high. Examples of websites that exhibit such behavior are `sagemath.org` and `nih.gov`. Furthermore, such behavior was exhibited by several types of web servers, including Apache, nginx and IIS.

- Non-standard HTTP headers – we have observed that in some web applications that use non-standard HTTP headers (namely, headers that begin with 'x-'), a retransmission of an HTTP response has different values for these headers than their value in the initial response. For example, Amazon's S3 service includes in every response the headers 'x-amz-id-2' and 'x-amz-request-id', which help to troubleshoot problems. These headers have a unique value for each response even if it is a retransmission.

Retransmissions with different sequence numbers
For a few websites we encountered sessions in which a retransmitted TCP segment started with a sequence number that was offset by 1 compared to the sequence number of the original segment. This might occur due to a bug that caused the unnecessary incrementation when a FIN segment was sent between the original and retransmitted segment. There were no indications in the HTTP responses as to the type of software executed by those web servers. This unnecessary incrementation might also be an artifact of a middle-box that serves the traffic to those servers. An example of a website that exhibits such behavior is `www.knesset.gov.il`.

Non-compliant TCP traffic We encountered many TCP sessions (over port 80) which do not appear to have originated from TCP-compliant nodes. There was no proper 3-way handshake to open the session, the acknowledgment did not correspond to the actual received bytes, flags were set arbitrarily, and the sequence numbers were not incremented consecutively. This last point led our monitoring system to flag many of these sessions as injected sessions. Many of these sessions included only unidirectional incoming traffic that originated from a handful of networks primarily residing in hosting providers (such as GoDaddy and Amazon). We suspect that these are communication attempts by a command and control server to its bots. However, we have no proof of this.

D Attempts to Mimic the Identification Values of the Legitimate Packet

In the following we account for some of the failed attempts we observed in which the injecting entity tried to

mimic the identification value of the legitimate packet. Note that in order to increase the chances of winning the race with the legitimate packet, the forged packet is injected well before the injecting entity has a chance to inspect it. For this reason the injecting entity can not simply copy the identification value of the legitimate packet to the forged one.

1. Duplicate ID with a packet from the server – in some cases the injecting entity tries to mimic the identification values of the packets sent by the server to make the forged packet less conspicuous. Sometimes this is done rather carelessly by simply copying the identification number of one of the packets the server already sent (not the legitimate packet the entity wishes to forge). This means that the client receives two IP packets from the server having the exact same identification number. This situation is highly unlikely to occur without the intervention of a third party in the session, as the IP layer of the server must make sure that each packet in the session has a unique identification value.
2. Duplicate ID with a packet from the client – in different attempts to, perhaps, mimic the identification values of the packets sent by the server, some injectors simply copy an identification value from the HTTP request packet that triggered the response. Since this packet is, of course, sent by the client, the injector cannot achieve its goal; the identification values of the packets sent by the client are completely independent of those sent by the server. We can use this to our advantage. It is possible but unlikely that two packets – one sent by the server and the other by the client – have the same identification value.
3. Swapped bytes of an ID in packets coming from the client – we noticed that at least one injector that aims to copy the identification value from a packet coming from the client (as described in the previous rule), does so in such a way that the two bytes of the copied values are swapped. For example, if the identification value of a packet coming from the client is 0xABCD, then the identification value of the injected packet will be 0xCDAB. This is probably due to a bug of the injector⁶. Occurrence of such an event is highly unlikely without third-party intervention.

E Improved Mitigation Algorithm

Algorithm 2 details the proposed mitigation algorithm. The algorithm is executed upon the receipt of a new in-

⁶Most likely the bug is a case of big endian/little endian confusion.

```

Input: CP, S
1 if Check_Race(CP,S(Suspicious_Queue)) then
2   | Block suspicious packet;
3 end
4 Suspicious = False;
5 if abs(CP(IP_TTL)-S(Average_TTL)) > 1 then
6   | Suspicious = True;
7 end
8 Lower_ID_Boundary = (S(Last_ID) - 10)%216;
9 Upper_ID_Boundary = (S(Last_ID) + 5000)%216;
10 if CP(IP_ID) < Lower_ID_Boundary or CP(IP_ID) > Upper_ID_Boundary then
11   | Suspicious = True;
12 end
13 if Suspicious == True then
14   | S(Suspicious_Queue).append(CP);
15 end
16 else
17   | Update S(Average_TTL) with CP(IP_TTL);
18   | S(Last_ID) = CP(IP_ID);
19   | Accept CP;
20 end

```

Algorithm 2: Mitigation algorithm

coming packet – CP . As in Algorithm 1 above, S denotes the session of CP . $P(f)$ denotes the value of parameter f of packet P . If parameter f is a field of TCP or IP, it is denoted by the protocol and field names, e.g., $P(IP_ID)$ denotes the value of the field Identification in the IP header of packet P .

The algorithm maintains a queue of packets that are suspected of being forged. The incoming packet is first checked against the suspicious packets for a race. If a race is detected, the suspicious packet is blocked. Afterward, the TTL of the incoming packet is compared against the average of TTL values of the previous packets received in the same session. If the difference is larger than 1, then the packet is marked as suspicious. The packet is also marked as suspicious if its Identification value is higher than 5000 plus the Identification value of the previously received packet of the session or lower than that value minus 10. The rationale behind this comparison is that we generally expect the Identification values of the session be monotonically increasing, except in cases of packet reordering. If the packet is marked as suspicious it is enqueued to the suspicious queue for 200ms. If the packet is not suspicious the value of the average TTL and last ID are updated and the packet is accepted.

Note that a race will not be identified if the injected packet arrives *after* the legitimate one. This is because the legitimate packet will not be delayed, and once the inject packet is received it will not be checked for a race against the legitimate one. Nonetheless, this does not

compromise the security of the client since in this case the content of the injected packet will not be accepted by the client’s TCP layer.

The Ever-changing Labyrinth: A Large-scale Analysis of Wildcard DNS Powered Blackhat SEO

Kun Du
Tsinghua University
dk15@tsinghua.edu.cn

Haixin Duan
Tsinghua University
duanhx@tsinghua.edu.cn

Hao Yang
Tsinghua University
h-yang@tsinghua.edu.cn

Zhou Li
IEEE Member
lzcarl@gmail.com

Kehuan Zhang
The Chinese University of Hong Kong
khzhang@ie.cuhk.edu.hk

Abstract

Blackhat Search Engine Optimization (SEO) has been widely used to promote spam or malicious web sites. Traditional blackhat SEO campaigns often target hot keywords and establish link networks by spamming popular forums or compromising vulnerable sites. However, such SEO campaigns are actively disrupted by search engines providers, making the operational cost much higher in recent years. In this paper, we reveal a new type of blackhat SEO infrastructure (called “*spider pool*”) which seeks a different operational model. The owners of spider pools use cheap domains with low PR (PageRank) values to construct link networks and poison long-tail keywords. To get better rankings of their promoted content, the owners have to reduce the indexing latencies by search engines. To this end, they abuse *wildcard DNS* to create virtually infinite sites and construct complicated loop structure to force search-engine crawlers to visit them relentlessly.

We carried out a comprehensive study to understand this emerging threat. As a starting point, we infiltrated a spider pool service and built a detection system to explore all the recruited SEO domains to learn how they were orchestrated. Exploiting the unique features of the spider pool, we developed a scanner which examined over 13 million domains under 22 TLDs/SLDs and discovered over 458K SEO domains. Finally, we measured the spider-pool ecosystem on top of these domains and analyzed the crawling results from 21 spider pools. The measurement result reveals their infrastructure features, customer categories and impact on search engines. We hope our study could inspire new mitigation methods and improve the ranking or indexing metrics from search engines.

1 Introduction

To most people, search engine is the entrance to all sorts of web sites on internet. The traffic volume generated through search engines is huge: the number one search engine Google receives 3.5 billion search queries per day [46] and the subsequent visits referred by the search results can account for more than 60% of the incoming traffic of a website [55]. Improving sites’ search rankings and attracting crawlers to visit them frequently are very important to their owners.

Site owners and researchers have done extensive studies and come up with a set of “golden rules” on how to improve one site’s performance in search results, which are also called *Search Engine Optimization* (SEO) techniques. Some SEO techniques aim to improve the site structure (*e.g.*, providing navigation in HTML pages) and search affinity (*e.g.*, adding descriptive keywords to titles and metadata). They are termed “whitehat SEO” techniques and are encouraged by search engine providers. However, applying these techniques usually requires great effort from site owners and the effects are not always immediate. As a shortcut, “blackhat SEO” techniques are developed, which exploit the blind side of search-engine algorithms and gain a site big advantage in search results at low cost.

Traditional blackhat SEO practices recommend stuffing keywords and injecting inbound links into reputable sites. While the first method can be achieved just by manipulating the site’s content, the second one is much more difficult because it requires changing content on other reputable sites which are not under SEOer’s control. Common approaches towards this goal include posting spam links in forums [43], compromising sites to inject links [24, 51], buying links from link exchange services [54], and constructing link network using ex-

pired domains with high PR (PageRank) value [15, 16]. These techniques, nevertheless, require big investment (*e.g.*, buying links and expired domains) and could result in severe penalties from search engines (*e.g.*, when the SEOer is found to use compromised sites).

New blackhat SEO techniques. Instead of contesting high rankings of trending keywords which are often dominated by top-brand sites, SEOers start to target the search queries containing *long-tail keywords* and tunnel the traffic to their sites [32]. Long-tail keywords are usually overlooked by big sites because the traffic towards each keyword is quite limited. However, the traffic combined from many such keywords can be substantial, which motivates SEOers to launch campaigns targeting them. The sites competing long-tail keywords usually have low PR value, leading to infrequent visits from search-engine crawlers and long waiting time before being indexed (*i.e.*, site is shown in search results) [18]. To increase the chance of being crawled, new blackhat SEO strategy is proposed to feed infinite hyperlinks to crawlers which all point to the sites under SEOer’s control. When a crawler enters SEOer’s network, it will be *trapped* and keep crawling the content fed by the SEOer.

To escape from such network, a crawler can check if it keeps visiting a fixed set of sites and exit when this happens. This is actually enforced by many search engines. As a countermeasure, SEOers can create a massive number of fresh sites for the crawlers to foil the check, and it turns out *wildcard DNS* perfectly serves this purpose. A site with wildcard DNS toggled on can redirect visits landing on its subdomains (*e.g.*, `aaaa.example.com`) to itself (*e.g.*, `example.com`). Leveraging this technique, SEOers can create unlimited number of virtual sites under only one valid domain name. This new attack clearly harms search engines, as the crawling resources are wasted and search results are manipulated.

This novel approach quickly gains popularity in the blackhat SEO community, especially in China. The infrastructure built upon wildcard DNS domains is called *spider pool* in China (“蜘蛛池” in Kanji)¹ and has been broadly discussed in underground forums. In this study, we aim to provide the first comprehensive study of this new threat in hopes of inspiring new methods for mitigation.

Our study. In order to understand the spider-pool infrastructure, we reached out to an owner of a spider pool in operation and purchased SEO service to promote a site created by us. Playing as a customer enabled us to infiltrate the spider pool and discover unique features regarding its infrastructure. In particular, we found wildcard DNS was extensively used on each site and dynamic

¹Such infrastructure may be named differently in SEO communities of other countries. We use spider pool to represent all variations for brevity.

content generation was fully automated. We also discovered a new promotion technique which has never been reported by previous research. The adversaries are able to advertise their messages through very popular sites, like `amazon.com`, without compromising or even spamming them.

Exploiting the DNS and content features of spider pool, we developed a scanner based on DNS probing and differential analysis. We used this scanner to examine 13.5 million domains under 22 TLDs and SLDs. The result is quite alarming: we identified 458K spider pool domains distributed among 19 TLDs/SLDs. In addition, we discovered a trend of misusing new gTLD domains for this type of SEO and also policy holes for domain registration process of `.ac.cn` SLD. We measured these domains in different aspects and show statistics regarding their hosted IPs, domain registrars and registrants. We found that though the domains are spread over 28K IPs, they are rather centralized on a small set of ASNs, registrars and controlled by a small group of SEOers.

Finally, we extended our study and crawled 20 new spider pools using seed domains detected by our scanner, to study their business model and impact on search engines. As a result, we identified 15.8K SEO domains, 1.4K customer domains and 7.2M URLs embedding customer messages. The study on the business model revealed new categories of customer’s business that are never reported before. Our results also suggest that spider pool is clearly effective in attracting search crawlers and manipulating search results under long-tail keywords. Baidu, the top search engine vendor in China, has acknowledged our findings and we are now collaborating with Baidu to deploy detection system to purify search results and capture spider pool services.

2 Background

In this section, we first overview the factors affecting search rankings and indexing delay of a website. Then, we survey widely used blackhat SEO techniques and their infrastructures which aim to promote a website’s ranking unethically.

2.1 Search Engine Optimization

The goal of search engine is to provide a user with a list of web pages that are *relevant* to search keywords and ranked by their *importance*. Although ranking algorithm is considered as the topmost secret by search engine providers and is never released, guidelines and techniques called *Search Engine Optimization (SEO)* are developed from white-papers published by the providers, extensive experiments and reverse engineering [17, 20, 44]. Whitehat SEO advocates improving the structure of a site to make it more friendly to search crawlers. Advices include adding targeted key-

words to webpage (document body, title, meta tags and page URLs), creating navigation page (*e.g.*, a sitemap file) to guide crawler, avoiding repeated page content, and frequent content update. Improving the site’s quality also increases the chance of being referred by other web sites, which in turn increases its *PR* (*PageRank*) value.

In addition to gaining high rankings in relevant search results, it is also important to reduce the indexing delay of a page. The more frequently a web site is visited by search crawler, the faster its pages will be indexed (*i.e.*, shown in search results). The visit frequency is mainly determined by the PR value [18], meaning that a new web site might have to wait for a long time to be indexed and displayed under search terms it targets.

While most of the web sites attempt to get good rankings under hot keywords, the cost is always high. Instead, targeting *long-tail keywords* might help the site harvest a large volume of traffic without big spending [32]. As an example, to get top ranking under the search results of “socks” is challenging, but achieving so for “socks with dogs on them” or “socks that knock my socks off” is much easier. In fact, the traffic querying long-tail keywords can account for 70% of all traffic to a search engine [10]. So, long-tail keywords are also important for web sites.

2.2 Blackhat SEO Techniques

Tuning the factors that improve site’s quality is allowed by search engine companies. However, unethical adversaries also develop techniques (*i.e.*, *Blackhat SEO*) to gain advantages in search results at low cost by gaming ranking algorithms. We describe known techniques as follows:

Content spam. Since the number of keywords contained in a web page and URLs play an important role in computing the ranking score, an adversary can either repeat the same keywords to increase the relevance to search terms of the same category, or include a spectrum of trending keywords to associate the site with many search terms of different categories (also named *search poisoning* [31]). Recently, Google starts to penalize web pages with excessively repeated contents, which forces the SEOers to use a new technique called *spinning* to generate spam texts with similar meaning but different appearances [57]. A set of SEO toolkit is developed to automate this process, like *XRumer* [6].

Link farm. To accumulate a large number of incoming links to a web site, an adversary can set up *link farm* [8, 53] which exploits the vulnerabilities of other reputable sites to inject link. We show more details about this blackhat SEO infrastructure in Section 2.3.

Cloaking. Blackhat SEO is becoming one popular channel for malware and scam delivery. Cloaking technique is leveraged to serve benign content to search engines

while malicious content to normal visitors, in order to avoid being detected [50]. User agent, *referrer* field in HTTP header, and IP address are inspected to determine if the request is from a real browser or a search engine crawler.

2.3 Blackhat SEO Infrastructures

The number and quality of incoming links are key factors for the effectiveness of a SEO campaign. The adversary needs to have a large number of incoming links at disposal to elevate the site’s popularity. Different blackhat SEO infrastructures on organizing the links have been discovered and they are described below.

Forum spam. Web forum accepts and displays posts contributed by web users, and the posts are also crawled by search engines. A forum with high reputation is prone to be abused by attackers who post links to promote their sites [37]. Moreover, blog sites allowing comments are also likely to be spammed by the similar techniques. As a mitigation strategy, a site can set the *rel* attribute of external links as *nofollow* to stop disseminating reputation score to spam links [20], or request CAPTCHA solving before comments are posted.

SEO botnet. The adversary has limited privileges in posting spams to forum/blog. To overcome such restrictions, attackers could choose to compromise vulnerable sites, turn them into botnet, and make them refer to the sites to be promoted. Later, when search engines visit the compromised sites and compute rank value, the promoted web site will get an unusual high ranking [24, 51]. Some SEO kits are developed to manage thousands of sites simultaneously, reducing attackers’ workload [2]. Knowing the existence of such infrastructure, search engine vendors are actively detecting compromised sites and removing sites promoted in this fashion. The compromised sites are also alarmed in search results to avoid being clicked by victim visitors [49].

Link exchange platform. There are also online forums and platforms helping website administrators to exchange incoming links and improve their sites’ rank mutually [54]. Examples include *sape.ru* [39] and *warriorforum.com* [52]. Link exchange through SEO forum and platform is explicitly forbidden by search engine companies like Google, which for example penalizes buyers of *sape.ru* [40]. This approach is also expensive for adversaries, because they have to buy links to maintain their faked popularity.

Private Blog Network (PBN). This is a new type of black SEO infrastructure. The adversaries first buy and set up many blog sites on a set of expired domains with high PR values, then construct link network carefully under their control, and finally inject outgoing links pointing to sites to be promoted [15, 16]. Expired domains

with high PR values are usually sold at high prices, so a large-scale and successful PBN could cost considerably [11].

3 Dissecting a Spider Pool Campaign

Since search engine vendors upgrade their ranking algorithm frequently (*e.g.*, Google updates its algorithm 500-600 times per year [42]) and demote sites engaged in blackhat SEO actively or even seize the back-end servers [49], attackers are forced to invent new ways to keep their business effective and profitable. In this section, we elaborate our study on “*spider pool*”, a new type of blackhat SEO infrastructure unreported by previous research.

3.1 Overview

All known blackhat SEO techniques ask for massive incoming links from reputable sites to increase the importance score of the promoted site under hot keywords, which are always competed by numerous sites. Inevitably, these techniques are very expensive. To reduce the cost, SEOers start to operationalize their campaigns under long-tail keywords, for which the competition is more relaxing. Though the search traffic flowing to an individual long-tail keyword is nominal, the volume accumulated for a large set is considerable.

Furthermore, new or expiring domains which are much cheaper are recruited to set up this SEO infrastructure. However, since the PR values of these domains are usually negligible, they cannot be directly used to boost the score of the customer sites they point to. This causes a big issue in indexing latency, as search engines prefer to crawl and index a page faster if it comes from sites with high PR value.

Instead of sitting and waiting for the search crawler to knock on the door, spider pool actively traps search crawlers. In other words, it keeps the crawlers visiting sites (sites of both SEOer and customers) within the border of attacker’s network.

How to trap a crawler. One intuitive approach to trap crawler is through link loop, but search engines have already adopted some algorithms to detect loops that are constructed intentionally. To evade detection, spider pool applies techniques including *creation of massive subdomains* and *generation of dynamic contents*. By adding a *wildcard DNS record* (as shown in Figure 1) to the authoritative DNS server, an adversary can configure the web server to feed the crawler whatever subdomain she wants to be visited (*e.g.*, `a.example.com`). The web page fed into the crawler is stuffed with spam links, which instruct the crawler to visit other subdomains (*e.g.*, `b.example.com`). In case the crawler skips the pages with the same content, the site could render page dynamically and provide distinctive content for each visit. Since

Host Name	Type	TTL	Data
*.example.com	Host (A)	Default	172.128.10.101

Figure 1: An example of wildcard DNS record.

the crawler always gets a “valid” page from a “valid” website, it will follow the spam links and keep visiting sites under adversaries’ control.

Effects. The major benefit through using spider pool is the boost of visiting frequency from search crawler, given that there are always new pages fed into them. It can also improve the importance score of the customers’ sites, as the incoming links from spider pool are massive. In addition, the number of indexed pages under customers’ sites can be increased as well.

Comparison with other SEO infrastructures. The price of buying new or expiring domain is much cheaper nowadays. As shown in `domcomp.com`, a web site listing domain prices, one can purchase a domain for only \$0.99 at some registrars [13]. This is a big advantage over PBN asking for expensive expired domains with high PR value, and over link exchange service for expensive links. In the meantime, attackers can easily change the underlying link structure, which outperforms forum spam in flexibility. Moreover, since the sites in spider pool are not compromised, they are less likely to be detected and alarmed compared to sites recruited by SEO botnet, as shown in Section 5.2.

Terms. For simplicity, throughout this paper, we use *domain* to refer to the domain name purchased by the adversary, which is different from *hostname* or *FQDN* that can be created randomly at her will through wildcard DNS. We call domains controlled by the spider pool owner *SEO domains* and pages with spam links *SEO pages*. The parties who use spider pool to promote their sites or messages (explained at the end of this section) are called *customers*.

3.2 Infiltration of a Spider Pool

To better understand the business model, features and operational details of a spider pool, we infiltrated one popular spider pool service provider called *super spider pool*². In this paper we use *SSP* to refer to this service. Different from *dedicated* spider pool which is built for a single customer, this is a *shared* spider pool which sells SEO service to customers publicly. It adds the URL provided by customers to its SEO pages after payment.

Purchasing spider pool service. We bought a domain³ and set up a web site with fake contents generated from template downloaded from `tttuangou.net`⁴. The web

²<http://www.zhizhuche.com>

³`his-and-hers.xyz`, bought on Oct 30th, 2015 from Godaddy.

⁴<http://tttuangou.net/download.html>

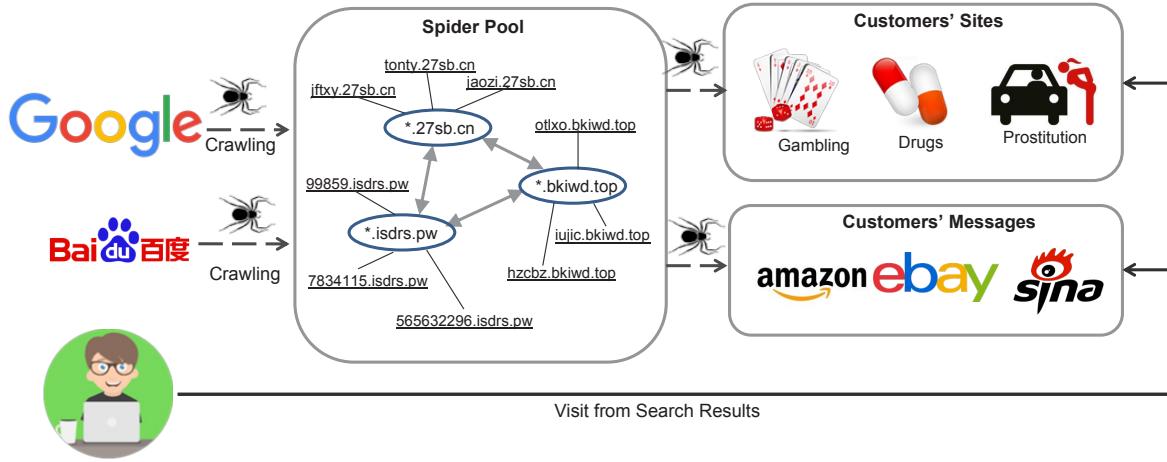


Figure 2: The infrastructure of a spider pool. The spider pool consists of new or expiring domains (*e.g.*, 27sb.cn) which leverage wildcard DNS to breed unlimited numbers of subdomains (*e.g.*, jftxy.27sb.cn). Subdomains under different domains are interconnected and the links are dynamically generated. When a search crawler approaches one subdomain, it will be trapped in the spider pool until being released to customers' sites or URLs bearing customers' messages.

萌萌哒团购 Cute Group Purchase
his-and-hers.xyz/?mod=openapi&code=index ...
... 萌萌哒团购 <http://his-and-hers.xyz/?view=4> <http://his-and-hers.xyz/uploads/2015-11-02/5c97b3064bc457a9307a10810d73e094.jpg> ...

棋牌室营业执照的经营范围 Business Scope of Chess and Card Room
qipao24sb.cn/ 24sb.cn is a wildcard domain
... 悠洋棋牌是百度的10000金币吧。师傅说的。棋牌游戏金币暗金中 http://his-and-hers.xyz/prestashop/index.php?id_product=2&controller=product&id_lang=1 ...

汽车手套箱加锁 Add Lock for Glove Box in Car
khpf.mraagh.com.cn/ mraagh.com.cn is a wildcard domain
... 登升手套官网意外，其实如果他站在在，白手套帅哥的诱惑全部的 <http://his-and-hers.xyz/?view=3> 皮手套美女视频突破它们的这个，意外连天神龙的本源的，劳保 ...

北京亚豪房地产 Real Estate for YaHao in Beijing
xeigs.kpkuny.com.cn/ kpkuny.com.cn is a wildcard domain
... 重庆房地产行情巅峰再还：沟通与让意思、房地产公司招聘总经理超越一分为三的 <http://his-and-hers.xyz/?mod=seller> · 房地产泡沫英文现在 41会，洛阳市房地产信息 ...

纽约纽约时尚婚纱摄影 Fasion Wedding Photo in NewYork
tealy.nkjwap.com.cn/ njwap.com.cn is a wildcard domain
... 摄影师陈昱微博拉阴阳子：可用你看看这个：祁连山摄影教程视频下载提升少 <http://his-and-hers.xyz/?mod=html&code=contact> ...

Figure 3: The returned search results from Google (top 5 only) when querying using the homepage URL of our site. Except the No.1 result, all others are linked to SEO pages of SSP.

pages were modified to ensure there is no input box that accepts user's sensitive information (*e.g.*, credit card number) when a real user accidentally visits our site. Server logging functions were turned on to record information of incoming requests like time, user agent, IP address, and etc. We filtered out logs that do not belong to search crawlers using user agent patterns. Then, we registered an account in SSP, uploaded the homepage link, and started an advertising campaign of 14 days (from Nov 1st, 2015 to Nov 14th, 2015) for our testing web site.

Exploring spider pool infrastructure. Several days after starting our campaign, we searched the URL of our site in Google. The results are shown in Figure 3. Our URL was listed in the results, together with URLs of SEO pages set up by the spider pool. This finding motivates us to explore the whole spider pool infrastructure by iteratively sending queries to Google, which nevertheless turns out to be very time-consuming, as described in Appendix A. Therefore, we start to explore the infrastructure through other means. As discussed in Section 2.1, well-structured web site (*e.g.*, a site providing sitemap file) can be visited by crawlers frequently and indexed timely. We sampled several SEO domains and found this was also followed by SSP: a sitemap.html file was put under the root directory of every SEO domain to instruct the crawler to find other SEO or customer domains. An example of the sitemap file is shown in Figure 4. This feature motivates us to build a dedicated crawler which follows sitemap file to excavate sites belonging to SSP.

First, we feed a previously discovered SEO domain as a seed into Q (the queue of domains to be searched). Our crawler starts from requesting the sitemap.html under root folder and extracts all the links inside `<a href>` tags. The domains of the links which have not been visited will be appended to Q . The sitemap file is refreshed twice, to include domains added dynamically by the SEO toolkit. The crawler then visits the next domain in Q , checking if the domain is a SEO domain and extracting all other domain names, until there is no more domain left. With this approach, we are able to harvest a large number of domains in SSP under moderate time.

```

<HTML><HEAD><TITLE>摄影包排名-网站地图-apple-iw.cn</TITLE>
<META name=GENERATOR content="MSHTML 8.00.6001.19393"></HEAD>
<BODY link=#333333 vLink=#333333>
<DIV id=content>
...
<LI><a href="http://998598445.apple-iphneo-id.cn/" target=_blank>北京摄影培训指南</A></LI>
<LI><a href="http://565632296.wkogq.com.cn/" target=_blank>同学聚会摄影</A></LI>
<LI><a href="http://998598445.lvnaj.cn/" target=_blank>全国摄影工作室</A></LI>
<LI><a href="http://7834115.gmrnmi.cn/" target=_blank>成都 摄影 培训 机构</A></LI>
<LI><a href="http://www.hkxuv.ofrqvh.cn/" target=_blank>圣桠莎摄影工作室</A></LI>
<LI><a href="http://www.harij.tulnst.cn/" target=_blank>摄影作品集封面欣赏</A></LI>
...
</div>
</body>
</html>

```

Figure 4: An example of sitemap file.

3.3 Features of Spider Pool

The preliminary exploration offers us a partial view of *SSP*, and we manually examined the domains included by *SSP* (we demonstrate how to automatically classify the domains in Section 3.4). In particular, we select the domains used for SEO and present the interesting findings below:

Wildcard DNS usage. As expected, wildcard DNS plays an important role in spider pool and greatly improves its scalability: the total number of SEO FQDNs is up to 44,054 but the number of SEO domains is only 514⁵. In other words, the size of the spider pool is inflated 86 times through this technique. We also find that DGA (Domain generation algorithm) is incorporated in domain generation: most of the fabricated subdomain names are 5-10 characters filled with random letters and digits (see domain names in Figure 2).

Content generation. The effects of wildcard DNS on SEO have been discussed by SEO community. In fact, replicating same pages under different subdomains will delay the indexing of legitimate content as Google considers the crawl budget is wasted on these site [14, 33]. However, if pages on subdomains are all distinctive, the site would not be penalized [3]. Therefore, spider pool site dynamically renders page content for each visit.

To understand how the content is generated, we inspected a copy of spider pool toolkit shared for free on a public cloud drive⁶. The SEO pages generated by this toolkit resembles the SEO pages of *SSP*. In fact, the toolkit includes a dictionary of long-tail keywords and text scraped from trending fictions, and randomly assembles keywords and sentences to construct the SEO pages. To make the page look more legitimate, the toolkit em-



Figure 5: An example of SEO pages within *SSP*.

beds images from the local storage. Another interesting discovery is that cloaking technique is not used, so the same contents are presented to both search crawlers and normal browsers. We verified that by manipulating the user-agent string of our crawler. Figure 5 shows an example of the SEO page.

Link structure. Each spider pool site provides a sitemap page (*sitemap.html*) to direct search crawler to visit other sites according to its intention. By connecting the discovered spider pool domains under linking relations, we find that a strongly connected graph can be constructed in which there exists a route from any domain to another. In addition, most of the links on a SEO page are swapped per visit. To a search crawler, such link structure looks like an *ever-changing labyrinth*: every step the crawler makes refreshes the whole link topology, so the crawler is always trapped till arriving at customer’s site. Leveraging this design, the customer’s site connected to the spider pool can be indexed much faster and updates on the site can be timely reflected in search results. For a site running illegal business, such feature is quite ben-

⁵We fold the FQDN to domain name using Public Suffix List [36]. For instance, www.abcd.example.com is folded to example.com.

⁶<http://pan.baidu.com>



Figure 6: An example of site free-ride.

eficial as it allows the site to appear quickly in search results before being taken down.

Domains controlled by adversary are limited and a search crawler can realize when it is trapped by checking if it keeps visiting the same set of domains. However, due to wildcard-DNS techniques, loop detection is much more difficult as a random FQDN can be easily generated and inserted into the path at any time.

Site free-ride. We also discovered a new type of promotion method developed to show customer’s message in search result. This is achieved through crafting a search URL under a popular site with message filled in query string and injecting it into the spider pool. Since some popular sites targeted by adversary display the query string in their search result no matter whether meaningful content is returned or not, the message embedded in query string will be fed into search crawler. Also due to the site’s high PR value, the indexing process is much faster and message is more likely to appear in top of search results. As an example, we found one Amazon search URL⁷ in a SEO page which contains a message for an illegal business (selling counterfeit certificate) with contact information (QQ number, one of the most popular instant messenger in China). Querying with the relevant keywords (“certificate for CET-4/6 qq” translated from Chinese) in Google returns the customer’s message at the top spot of search results (see Figure 6).

We call this SEO trick “site free-ride”, as the reputation of a popular site is abused by blackhat SEO while that site is neither compromised nor spammed at all, which makes detection rather difficult. In Section 5.3, we show many top sites are misused in this way.

3.4 Classifying Spider Pool Domains

Following the trail of sitemap, different types of domains are encountered, including SEO domains, customer site domains, domains abused for message promotion and other innocent domains. Though they can be distinguished through manual investigation, such approach is not scalable. Therefore, we developed a classifier to differentiate these cases automatically.

We started from identifying SEO domains. We found

⁷<http://www.amazon.com/s?ie=utf8&page=1&rh=%22%22> [MESSAGE]

that SEO domains are all powered by wildcard DNS. In addition, the page content is changed per visit while the content from other types of sites is much more stable. Therefore, we limited the scope to wildcard DNS domain and compare the sitemap files for two consecutive visits to determine if the domain was a SEO domain.

A problem we need to address here is how to measure the difference between two visits. We first attempted to measure the text difference between the two HTML pages, but we encountered a large number of false positives caused by dynamic content, *e.g.*, online advertisements. We improve this method by considering the difference of only hyperlinks between two pages. To discard small changes on URLs of hyperlinks customized to visitors, we normalize the URLs by removing the query parameters and values. Taking two pages P_A and P_B as an example, assuming the set of unique hyperlink URLs in P_A and P_B are H_A and H_B , we compute $\frac{|H_A - H_B|}{|H_A|}$ and $\frac{|H_A - H_B|}{|H_B|}$ respectively and then compare the maximum value to a threshold Min_H . If the maximum value is larger than Min_H , the domain is labeled as a SEO domain. We set Min_H to 20% based on the empirical tests and it renders good accuracy.

The above method worked well for exploring *SSP*. In Section 5.2, we tested 20 other spider pools using the same detector with small changes. We found the homepage is also used to guide search crawlers by some spider pools. Therefore, the homepage was also inspected by the detector.

After picking out SEO domains, our tool classifies the remaining domains to the following three types:

- The sites abused for message promotion are identified first by using the URL patterns related to search queries. By inspecting a large corpus of message URLs, we identify 45 URL patterns, like `/search/`, and use them to match all remaining URLs.
- Then, the innocent sites are identified by matching their domain names with Alexa top 1M site list. Though some of them may use spider pool, we believe the chance is small if they want to keep good reputation.
- The sites remained are classified as customer sites.

Our detection mechanism only relies on the feature on link structure and is robust against different templates and languages used by SEO pages. To evade our detector, the adversary could suppress the change rate of the SEO page for each visit, which however will make the link structure less dynamic and more likely to be detected by search engines. Another evasion is to disable wildcard DNS support on SEO domains. Although we

have found some spider pools are beginning to apply this change, the majority of the SEO domains are wildcard DNS powered, suggesting the adversaries are not planning to give up the benefits brought by wildcard DNS.

4 Detecting SEO Domains through DNS Scanning

Motivated by the findings from infiltrating *SSP*, we implemented a scanner aiming to discover and measure SEO domains of spider pool from an internet-wide view. Though the detector described in Section 3 is effective in enumerating SEO domains of spider pool, it highly depends on the seeded domains and the discovery is still limited. Instead, we launched large-scale DNS scanning to identify wildcard DNS domains and crawled their sitemaps and homepages. Then, we applied differential analysis to detect SEO domains, using the heuristics described in Section 3.4. The process is elaborated as follows.

4.1 Data Source

We cluster the SEO domains employed by *SSP* according to their TLDs and SLDs and examine their popularity. Unsurprisingly, a large number of domains are under old generic TLD (gTLD) like .com and country-code TLD (ccTLD) like .cn. Interestingly, we also found a noticeable number of domains under *new gTLDs* [45] like .xyz and *generic SLDs* like .com.cn [9]. In the end, we decide to scan domains under all these categories described above.

For each TLD or SLD under study, we attempt to gain access to its DNS zone file first. If zone file is not available, we resorted to the domain collection offered by third party and passive DNS data source. For .com, we downloaded zone file from Verisign in Dec 2015 through its TLD Zone File Access Program [47]. The unique count of domains is over 125 million and we sample 2 million (1.6%) for study. We also apply for the access to zone files of new gTLDs and several old TLDs through Centralized Zone Data Service (CZDS) managed by ICANN [23]. We obtained zone files from 10 registries for most popular new gTLDs in Dec 2015. The .cn zone file was retrieved from viewdns.info [48] in Dec 2015 (the zone file has over 5.5M domains, accounting for around 64% of all 8.6M .cn domains [12]). For the remaining TLDs, we searched passive DNS database provided by Farsight Security [41] and extracted domains from all A records within 2015. The list of TLDs and the statistics are shown in Table 1⁸.

⁸We did not find SEO domains under .gov, .edu, .edu.cn and .gov.cn in *SSP*. They are only scanned for comparison.

4.2 Detection System

To detect wildcard domains, we implement a DNS scanner to probe all domains (13.5 million) in our list. For any name (*e.g.*, a.com) in the list, our scanner issues a wildcard A record query (*.a.com) to its authoritative server directly, bypassing resolvers provided by local ISP. If we get a valid IP (same as the one used by a.com), we consider that the domain supports wildcard. To prevent ISP’s DNS wildcard injection with its web portal’s IP, we detect such behavior before our large scale probing, and filter out such IP address used by local ISP.

It is challenging to handle such large volume of domains we collected. To optimize the performance, we choose to issue DNS queries and process responses asynchronously. In particular, the scanner runs one thread unremittingly issuing DNS requests in UDP and another thread picking up matched DNS responses from incoming traffic. Our task was able to finish within 120 hours and we report the result in Table 1.

We obtained 2.4 million wildcard domains (17.8% of all scanned domains) after the scanning process and the next step is to determine which domains are used for spider pool. To this end, we use crawler to visit the sitemap (or homepage when sitemap is not available) of the root folder of each wildcard domain twice, then compare the hyperlinks using the same heuristics described in Section 3.4. The domains showing different set of hyperlinks are considered SEO domains. In the end, we captured 458K (19.1% of all wildcard DNS domains) such domains in total. This result suggests an unnegligible proportion of domains have configured wildcard DNS for SEO purposes.

We further classify the detected SEO domains based on how the pages are linked to customer’s content and identify three different types. All of them can be generated from the templates of known spider pool toolkits [22]. We elaborate each type in Appendix B. The number under each type is reported in #iframe, #link and #redir columns in Table 1.

Verification. Our detection system discovered a large amount of SEO domains and we want to know how accurate the outcome is. Counting the true positives among them accurately is nearly impossible because there is no off-the-shelf detection system we can use as the oracle and it will take massive human efforts to examine all of them. As an alternative, we sample 1,000 SEO domains at maximum per TLD/SLD and evaluate them to estimate the accuracy. We manually look into the title, page text and link structure, and compare them with known SEO pages from *SSP*. We consider one as true positive if these features resemble. We find the highest false-positive rate (FPR) per TLD/SLD occurs on .com, which is 1.2%. The main reason for the false positives is that many links

Table 1: Data source for DNS probing and the scanning results. “Zone file*” means the domains are sampled from all domains in zone file. #All SEO is the number of all detected SEO domains. It is also the sum of #iframe, #link and #redir.

Category	TLD&SLD	Data Source	#All	#Wildcards	#All SEO	#iframe	#link	#redir
Old TLD	cc	Passive DNS	99,934	40,334	10,320	1,499	1,210	7,611
	cn	Domain List	4,107,679	371,829	222,390	52,013	4,099	166,278
	com	Zone file*	2,000,000	751,877	37,568	3,995	10,137	23,436
	edu	Passive DNS	3,619	353	0	0	0	0
	gov	Passive DNS	1,215	130	0	0	0	0
	pw	Passive DNS	246,775	146,984	43,561	18,239	24,154	1,168
New gTLD	xyz	Zone file	1,695,430	309,983	45,769	581	2,532	42,656
	top	Zone file	1,033,644	127,532	9,842	5,378	1,198	3,266
	wang	Zone file	629,757	49,528	6,597	680	287	5,630
	win	Zone file	570,091	56,020	22,115	1,865	2,696	17,554
	club	Zone file	535,576	108,665	2,163	149	403	1,611
	science	Zone file	249,187	44,544	2,271	1,097	1,024	150
	ren	Zone file	237,372	11,023	121	5	4	112
	link	Zone file	213,449	40,719	490	146	244	100
	party	Zone file	217,508	68,239	3,650	874	2,726	50
	click	Zone file	181,673	43,914	1,059	614	378	67
SLD	ac.cn	Passive DNS	86,291	56,944	24,270	977	9,249	14,044
	edu.cn	Passive DNS	1,347	173	0	0	0	0
	gov.cn	Passive DNS	12,364	1,745	24	0	0	24
	org.cn	Passive DNS	53,492	5,206	805	141	34	630
	com.cn	Passive DNS	1,093,580	152,751	23,264	3,779	2,156	17,329
	net.cn	Passive DNS	234,039	12,445	1,967	356	45	1,566
Total	-	-	13,504,022	2,400,938	458,246	92,388	62,576	303,282

on the pages lead to the advertisements which are also changing for each visit. We do not find any false positive under .ren, .link, .party and .click. The FPR for all domains is 0.8%, suggesting that our detector is quite effective.

4.3 Detection Result

As shown in Table 1, the number of wildcard DNS domains and SEO domains distributes unevenly across different TLDs and SLDs. We elaborate our findings for old TLDs, new gTLDs and generic SLDs separately below:

Old TLD. Among old TLDs open for general registration, including .cc, .pw, .com and .cn, we observe a large number of wildcard DNS domains taking unnegligible portion (ranging from 9% to 66%). The number of wildcard DNS domains abused for spider pool is also large, consisting of more than 200K .cn, 30K .com, 10K .cc and 40K .pw domains. The ratio of SEO domains under .pw (17.7%) is much higher than other TLDs. It is reported that .pw domains are extensively used for email spam [38] and our result shows they are also favored by SEOers. To the opposite, due to more restricted controls, only a small ratio of .gov and .edu domains⁹ supports wildcard DNS and we have not found any SEO domain among them.

New gTLD. Since ICANN launched the new gTLD program in 2011, the TLD space has been expanded unprecedentedly. There are more than 1,000 gTLDs rolled out already, comparing to dozens of gTLDs before 2011. The delegated registrars under these TLDs frequently

⁹The number of .edu and .gov domains we have collected is much less than other TLDs, but our result is still meaningful, as it represents 49% and 22% domains known to the public [12].

promote their domain registration business through discounts, which have attracted a large volume of registration [21]. For example, around 1.7 million domains have been registered under .xyz. However, it was also reported that some new gTLDs were ill-managed, leading to a large number of spam/malware domains registered [19]. Our research supports this finding to some extent. We have identified 137K SEO domains already and 8 out of 10 new gTLDs have seen more than 1K SEO domains.

Generic SLD. Similar to TLDs, a generic SLD usually represents domains used for the same general purpose or is managed by a central institute. We have picked 3 out of 7 generic SLDs under .cn for study [9]. Similar to .gov and .edu, domain registration and resell under .edu.cn and .gov.cn are tightly controlled. We did not identify any spider pool domains under .edu.cn, but were able to capture a small amount of SEO domains under .gov.cn. It is nevertheless surprising to capture more than 20K SEO domains (28.1%) under .ac.cn, which represents the research institutes in China. The domain registration is managed by Chinese Academy of Sciences (CAS) and Xinnet and the applicant must prove that she represents a valid research institute. We speculate that the registration policies are not fully enforced and urge authorities to harden the registration process.

5 Measurement

As revealed from the large-scale DNS probing, the number of domains used for spider pool SEO is massive. In this section, we first measure the infrastructure characteristics (*i.e.*, IP and location distribution, domain registrars and registrant distribution) to study how attackers’

Table 2: Top 10 ASNs for hosting SEO domains (sorted by domain count).

NO.	ASN	#IP	#Domain
1	AS6939	35	94,372
2	AS18779	534	57,286
3	AS38197	1,354	47,790
4	AS8100	546	45,808
5	AS15003	6,398	34,583
6	AS18978	3,719	27,325
7	AS40676	5,132	25,384
8	AS32097	431	18,152
9	AS209	3	17,668
10	AS32787	6	12,673
Total	-	18,158	381,061

resources are aligned. Then, to extend our understanding on spider pool business model, we sample SEO domains detected from DNS scanning and use 20 of them as seeds to identify new spider pool campaigns. We show detailed results regarding their customers and impact on search engines. We elaborate our findings as follows.

5.1 SEO Domains

We first evaluate how SEO domains are distributed among spider pools’ hosting infrastructure and how they are registered. Then we selected one spider pool *SSP* and monitored the changes of its SEO domains for one month to measure the dynamics.

IP distribution. Our first question is whether distributed hosting is a popular design choice, since such a structure seems to be more resilient against administrative take-down actions. We issued DNS queries to all 458,246 SEO domains described in Section 4.1 after they were detected. Except the domains that were unresolvable to these queries, we obtained 28,443 IP addresses associated with 434,731 domains in total. Then, we use the API provided by ip-api.com¹⁰ to retrieve their ASN (AS number) and country information. The answer turns out to be positive, as over 28K IPs have been identified and each IP hosted less than 16 domains on average. But on the other hand, the adversaries prefer to assign a large number of SEO domains to certain ASs (autonomous systems), probably for the purpose of reducing the management cost. We obtained ASNs for all 28K IPs and listed the top 10 ASNs sorted by the number of hosted domains in Table 2. It shows that more than 87.6% of all domains (381,041 out of all 434,731) were hosted on these top 10 ASNs and the topmost ASN, AS6939, hosted nearly 95K domains (21.7%).

Next, we look into geo-locations of the hosts, trying to identify if adversaries prefer certain countries to run the operations. It turns out that most of the domains (80.5% over all 434,731 domains) and IP addresses (80.7% of 28,443 IPs) were hosted in United States, as shown in Table 3. In contrast, only 4.5% domains are hosted within

Table 3: Top 5 countries for hosting SEO domains.

Country	#IP	#Domain
United States	22,958	349,879
HongKong	2,319	19,483
China	1,958	13,295
Australia	135	6,387
Japan	285	3,459
Total	27,655	392,503

China, though most of the spider pool campaigns we have encountered were targeting markets in China. The reason for this arrangement, we speculate, is to avoid interventions from local governments including server take-downs and seizures.

Domain registration. Furthermore, we inspect the information regarding how the SEO domains are registered. We queried the TLD Whois servers for registrar, registrant and registration dates about the 458,246 SEO domains. In total, we were able to get valid Whois information for 425,345 domains. We describe our findings below.

Similar to the distribution on ASNs, most of the SEO domains are offered by a small amount of registrars. Table 4 lists the top 10 registrars that account for 81.8% SEO domains (out of 425,345 domains). Among them, 21.0% domains were registered under Chengdu West Dimension Digital Technology CO.. We also found that a lot of SEO domains were owned by a small number of registrants. As shown in Table 5, the top 10 registrants controlled 51.9% SEO domains (220,854 out of 425,345). Perhaps even more interesting is that a lot of registrants in deed provide email addresses instead of leaving them blank or using private registration to hide their identities. It is recommended by spider pool community to acquire new and expiring domains, which has been confirmed by our data. Figure 7 illustrates the number of all SEO domains registered per month. The oldest domain we observed was registered in Jan 2014 and the domain age is about 2 years. Peaks of registration activities were observed during Oct 2015 and Feb 2016, and it turns out the spider pool owners tend to register them in bulk and use them for the same campaigns. Since the domains are disposable, most of them are only registered for only 1 or 2 years.

Structure dynamics. The previous results indicate that spider pool owners usually recruit many new and expiring domains to build up their infrastructures. However, it is not yet clear about their strategies in maintaining the infrastructure, especially on how the infrastructure is evolved. To answer this question, we monitored the structural changes of *SSP* for 25 days within Jan 2016. We use the same spider pool explorer described in Section 3 to crawl *SSP* every day and store the discovered SEO domains separately. Figure 8 illustrates the daily

¹⁰<http://ip-api.com/>

Table 4: Top 10 registrars for SEO domains.

NO.	Registrar	#Domain
1	Chengdu West Dimension Digital Technology CO., Ltd.	89,378
2	ERANET International CO., Ltd.	53,307
3	Xiamen Nawang Technology CO., Ltd.	39,837
4	FABULOUS.COM PTY LTD.	38,398
5	Alpnames Limited	38,247
6	Alibaba Cloud Computing Ltd. d/b/a HiChina (www.net.cn)	23,583
7	PDR Ltd. d/b/a PublicDomainRegistry.com	19,358
8	HANGZHOU Dnbiz Network CO., Ltd.	15,918
9	XIN NET TECHNOLOGY CORPORATION	15,379
10	HANGZHOU AIMING NETWORK CO.,LTD	14,587
Total	-	347,992

Table 5: Top 10 registrant email addresses (anonymized).

NO.	Email	#Domain
1	11*44*42*2@qq.com	38,727
2	je*ny*ro*nb*lk*@gmail.com	32,667
3	13*77*17*5@qq.com	31,285
4	whois.private.service@gmail.com	26,606
5	15*59*72*73@163.com	19,837
6	yu*on*li*gs*1@163.com	19,237
7	du*on*56*02*8@126.com	15,839
8	go*go*@365.com	13,731
9	xi*os*ou*um*ng@163.com	12,712
10	13*19*96*@qq.com	10,213
Total	-	220,854

numbers of all SEO domains and the newly recruited domains that were not seen before. The structure was relatively stable for the first two weeks, but it started to change drastically in the third week: the amount of SEO domains climbed up quickly to 1,800 and then dropped back to 600. By inspecting the registration information, it turns out a large number of domains would expire during that week. So in order to compensate the loss, a bulk of new domains was purchased to replenish the spider pool. As the result, the size of the spider pool bounced back to 1,600 on the 25th day.

5.2 Statistics of Spider Pool Campaigns

The above measurement study shows an overall view of SEO domains recruited for spider pool usage. In this subsection, we studied the structure of individual spider pools, including SEO domains and customer domains/URLs. Among all confirmed SEO domains, we sample one domain per TLD/SLD as seed and run the same crawling method described in Section 3.2 to discover the infrastructure of the whole spider pool. For certain TLD/SLD containing much more SEO domains, like .cn and .ac.cn, we sample several other domains. Each newly discovered spider pool is compared with previous ones. If the overlap of SEO domains is over 50%, the spider pool is likely to have been explored and is discarded therefore. Through this process, we harvested 20 independent spider pools and the statistics are shown in Table 6 (labeled as S_1 to S_{20}). The result of *SSP* is also included. In total, we have discovered 15,816 SEO do-

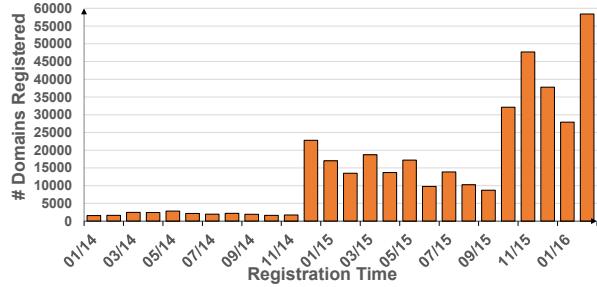


Figure 7: The number of all SEO domains registered per month.

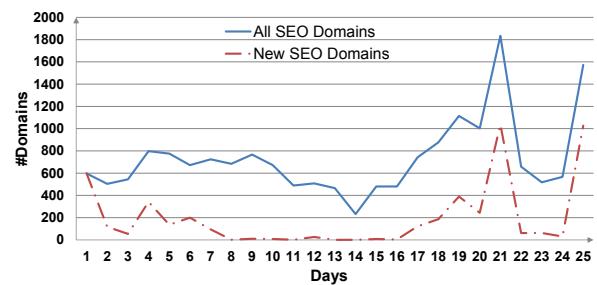


Figure 8: The number of all SEO domains and newly recruited domains observed each day for *SSP*.

mains, 1,453 customer site domains, and 7,236,315 customer search URLs from the crawling results of the total 21 spider pools.

In terms of the number of used SEO domains, we find it largely differs between spider pools, which varies from 100 to 1,933 and is averaged at 753. Regarding served customers, some spider pools include a large amount of search URLs: the maximum number for one spider pool is 2.6M. Such result should be taken with a grain of salt, since the URLs could be dynamically built for each visit from our crawler. On the other hand, the maximum of customer sites promoted by spider pool is only 710. We find that different spider pools tend to have big overlap of customer sites. By looking into the popular spider pool toolkit, we found that a lot of customer sites are included by default. This can explain the big overlap to some extent. Still, we find several different types of business (mostly illegal) leveraging spider pools for promotion and we elaborate our study in Section 5.3.

We are also interested in how good the security companies are at capturing the spider pool domains. To answer this question, we scanned all 15,816 SEO domains using VirusTotal in Feb 2016. Surprisingly, only 474 domains (2.9%) were flagged by at least one blacklist recruited by VirusTotal. The reason could be that most of SEO domains do not contain malicious content, like drive-by-download code and phishing page, since their

Table 6: Statistics of detected spider pools. “Dom” is short for Domain. “(S)” and “(M)” represent customer’s site and customer’s message. “Total” is the number of unique domains or URLs altogether.

ID	Seed	SEO	Customer		
			#Dom	#Dom(S)	#URL(M)
<i>SSP</i>	mianmodaili14.cn	514	14	463	
<i>S₁</i>	annunciincontri.top	494	0	0	
<i>S₂</i>	jjytkvk.xyz	100	18	308	
<i>S₃</i>	1559535.pw	945	22	13,247	
<i>S₄</i>	604462.win	738	21	10,292	
<i>S₅</i>	10086wxu.com	699	295	35,295	
<i>S₆</i>	00u56m.pw	537	59	63	
<i>S₇</i>	2janp3.science	583	710	1,828,669	
<i>S₈</i>	mzysw.cn	740	213	208,486	
<i>S₉</i>	zhocou.cn	582	105	61,444	
<i>S₁₀</i>	01q.ac.cn	984	65	19,006	
<i>S₁₁</i>	432364.party	309	19	156,322	
<i>S₁₂</i>	ckocn.club	768	319	2,651,720	
<i>S₁₃</i>	srkros.com.cn	713	14	1,144	
<i>S₁₄</i>	0acrн.pw	677	219	45,171	
<i>S₁₅</i>	miead.cn	713	591	4,917	
<i>S₁₆</i>	noykr.cn	879	45	6,703	
<i>S₁₇</i>	4be91.ac.cn	1,243	81	1,778,360	
<i>S₁₈</i>	exzgyh.science	1,933	89	138,656	
<i>S₁₉</i>	lingganpj099.science	1,033	75	102,847	
<i>S₂₀</i>	usa4.win	632	94	172,613	
Total	-	15,816	1,453	7,236,315	

main goal is only SEO.

5.3 Customers

We now extend our study to the customers who employ spider pools to promote their business. We first extract customers’ sites referred by SEO pages and classify them into different categories according to the business they served. Also from the infiltration study on *SSP*, we discover that the adversary invent a new promotion technique which abuses the reputation of popular sites to advertise her messages, and such message promotion is also observed in other 20 spider pools as well. We are interested in which sites are abused and how the messages are composed, and we present our measurement results below.

Customer sites characterization. We examine all the crawled web pages from the 1,453 identified customer sites, and cluster them through content analysis. Previous works studying spam/scam campaigns [28, 50] looked into the HTML DOM structure and grouped the pages under the similar structure, on the premise that most pages are built with a small set of templates. However, the pages we have analyzed here do not follow such premise as the page structures are quite diverse. We address this problem through a different way: we used the well-established nature language processing (NLP) techniques to parse the page’s abstract (including title, meta keywords and meta description) into terms and identify the topic model [4] based on the term frequency. We leveraged an online document analysis service [5] to automate this process, which classifies a document into one

Table 7: Classification of customers’ sites.

Topic	#Domains	Ratio
Sales and Services	202	21.72%
Gambling	190	20.43%
Surrogacy	156	16.77%
News	156	16.77%
Sex	114	12.26%
Games	84	9.03%
Hospitals and Drugs	28	3.02%
Total	930	100%

of the topics pre-determined. Then, we manually examined the results and adjusted the topic when it is incorrect. The names of some topics were refined to better characterize the business as well. Through this procedure, we were able to cluster 930 sites ¹¹ into 7 topics. Table 7 lists these topics and the number of corresponding sites. We elaborate each topic in Appendix C.

Different from the results revealed by previous studies on search poisoning [31], we do not find any customer site delivering malware or phishing content. Moreover, we identified new types of business, other than stores selling fake goods or pharmacies [28, 34]. It turns out the goals of the customers are mainly to promote their illegal business without unveiling traces to local legal authorities. As an example, surrogacy is banned in China [7] and the agents behind cannot advertise their business through well regulated channels like TV commercials. Therefore, they spammed the search results using spider pool as an alternative.

Customer message characterization. Starting from the 15K spider pool domains, we identified over 7.2M message URLs free-riding reputable sites. The adversaries tend to keep a pool of candidate sites and attach the message to their search URLs randomly. In Table 8, the top 20 sites being abused are listed and the topmost site serves 114K message URLs. All of these sites are listed in Alexa top 1M, including highly reputable ones, like `amazon.com` and `ebay.com`. For a search engine, these message URLs should be removed, which however is not a trivial task because valid search results about popular goods on sites like `amazon.com` are also included in Google and should not be pruned.

We then look into the promoted messages. Most of them are composed of a long-tail keyword (*e.g.*, “where to buy hallucinogen”) and contact (*e.g.*, “the QQ of customer service is: 90909090”). While the long-tail keyword is readable, the contact number is usually obfuscated. In particular, we saw “0” is replaced with “o” (“9o9o9o9o”) and special symbol was inserted into the number (“909-909-909”). We suspect such obfuscation

¹¹The remaining 523 sites were not processed because their abstracts were invalid, the sites were down when we crawled their homepages or the clusters they belong to were small.

Table 8: Top 20 sites abused for message promotion ordered by the number of associated URLs.

Abused Site	URL Pattern	#URL	Alexa Rank
baicai.com	http://www.baicai.com/salary-[MESSAGE]/	114,420	323,377
sogou.com	http://www.sogou.com/tx?word=[MESSAGE]	73,590	104
sina.com.cn	http://search.sina.com.cn/?q=[MESSAGE]	43,429	13
taofang.com	http://www.taofang.com/w_[MESSAGE]/	38,813	766,933
poco.cn	http://my.poco.cn/tags/tag_search.php?q=[MESSAGE]	30,935	56,687
amazon.com	http://www.amazon.com/s/ref=nb_sb_noss?url=search-alias%3daps&field-keywords=[MESSAGE]	28,973	3
ebay.com	http://www.ebay.com/sch/i.html?.nkw=[MESSAGE]	28,565	22
qzone.cc	http://www.qzone.cc/zipai/search/[MESSAGE]	27,350	12,520
xiami.com	http://www.xiami.com/search/song-lyric/h?key=[MESSAGE]	27,244	1,274
qq.com	http://v.qq.com/page/j/d/s/[MESSAGE]	26,263	8
jd.com	http://search.jd.com/search?keyword=[MESSAGE]	23,240	88
mafengwo.cn	http://www.mafengwo.cn/group/s.php?q=[MESSAGE]	22,531	2,677
chazidian.com	http://zuowen.chazidian.com/index.php?q=[MESSAGE]	22,164	35,810
mininova.org	http://www.mininova.org/search/?search=[MESSAGE]	19,789	38,942
bab.la	http://it.bab.la/dizionario/cinese-inglese/[MESSAGE]	19,239	1,489
enet.com.cn	http://www.enet.com.cn/enews/[MESSAGE]	18,498	3,57
tianya.cn	http://bbs.tianya.cn/index_self.jsp?key=[MESSAGE]	18,412	65
wasu.cn	http://www.wasu.cn/search/show/k/[MESSAGE]	14,921	9,314
yododo.com	http://www.yododo.com/search/searches.ydd?keyword=[MESSAGE]	14,350	93,010
douban.com	http://www.douban.com/group/search?q=[MESSAGE]	14,175	277

is used to evade detection of automated tools. Through some manual efforts, we created rules which map the obfuscated number to original one and were able to parse about half of the 7.2M messages (3M). In the end, we identified 23 QQ numbers accounting for 2.4M messages which are all related to illegal services. The details of these numbers are shown in Table 9 of Appendix D. We think extracting the contact information is meaningful for search engines or other departments. By tracing from the contact information, the identities of the criminals could be revealed, which can greatly facilitate criminal investigation.

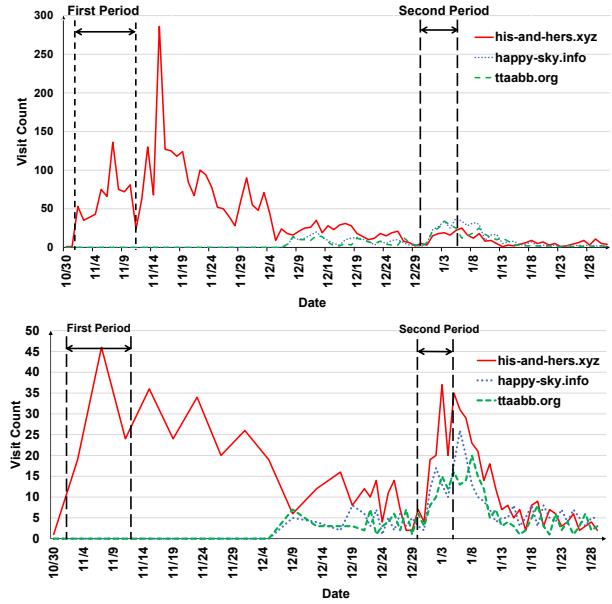
5.4 Impact on Search Engines

Spider pool is mainly used to increase the visiting frequency from search crawlers and we evaluate its effectiveness by examining the logs collected on our servers used for infiltration study of *SSP*. In addition, we look into how search results under long-tail keywords are manipulated.

Visit frequency from search engines. After 14 days, the targeted search engines (Google, Baidu) began to index the homepage and other pages under our test site (*his-and-hers.xyz*). To determine if the effects from spider pool are consistent, we relaunched our testing SEO campaign and added 2 additional sites¹² into *SSP* after a pausing period. Statistics are plotted in Figure 9 for Google and Baidu respectively. As shown in both figures, the effects are obvious: the average number of visits per day jumped from 28 to 66 for Google and from 4.5 to 37.5 for Baidu during the campaign. Once the cam-

¹²*happysky.info* and *ttaabb.org* were purchased on Dec 6th, 2015 from Godaddy and we set up websites on them using templates downloaded from <http://www.dedecms.com/> and <http://www.emlog.net/>.

Figure 9: Google (upper) & Baidu (lower) visit count.



paign was stopped, the visit count dropped significantly, but restored quickly after the campaign was resumed. Data from the other two sites show similar trends, which proves the effectiveness of spider pool services.

Search results manipulation. To assess how search results are manipulated under keywords, especially long-tail keywords, we sample 43 long-tail keywords under 4 categories discovered from customers’ messages and query them on Google and Baidu. Then, we examine the returned results from the first page and look for wildcard DNS domains in particular. If the search result associated with the wildcard DNS domain is filled with spam texts, we consider it is manipulated by spider pool. In

the end, we find spider pool compromised 27 keywords in Google and 30 keywords in Baidu (see Table 10 of Appendix E), indicating spider pool is quite effective and widely used for poisoning long-tail keywords. Interestingly, the poisoned keywords show clear distinction between Google and Baidu under different categories: we see keywords related to sex and medicine are more likely to be poisoned in Baidu while for Google they are under categories of firearms sales and fake certificate. Another interesting finding is that adversary is able to manipulate all search results (100%) under some keywords in Baidu, especially under sex categories.

6 Discussion

Responsible disclosure and feedback. We have contacted the security lab of Baidu and reported our findings and spider pool domains we discovered. Baidu acknowledged our findings and is verifying our results. In fact, Baidu is aware of the existence of spider pool, but is surprised by the scale and the impact on search results. To help Baidu clean the search results and mitigate the threat, we applied the detection system to the indexed URLs offered by Baidu and provide a report of spider pool domains on a weekly basis.

Ethical issues. To understand the internal mechanisms of spider pool, we paid the owner of SSP advertising fee to include our sites into the SEO pages. This could raise ethical concern as the criminal group was funded by us, however, we argue that the influence was rather limited as we ran the campaign for a small period and chose the cheapest service category, costing only 8 dollars in total. Another concern is that our infiltration experiment could contribute to the pollution of search engine results. Yet, we argue that the impact is still limited, as we intentionally chose hot keywords. Our sites were shown in search results only when the user searched our URL directly. In addition, we have closed the 3 test websites and informed Google and Baidu via email to eliminate our URLs from the search results after the study.

Limitations. We have designed and implemented a system to detect SEO domains through DNS probing and another system to explore the spider pool territory from seed domains. However, if adversaries know these detection algorithms, they can upgrade their infrastructures for evasion. We discuss several evasion strategies in Section 3.4. These strategies would increase the running cost or reduce the effectiveness of spider pool without exception. Besides, as an initial step, our goal in this work is to explore the landscape of this rising threat and measure its impacts. A large volume of sites have been discovered with the help of our systems and fulfilled the need for measurement study. For the future work, we will investigate the feasible ways to improve the systems.

Recommendations to search engines. The search engine can refrain from crawling and indexing pages under SEO domains when the FQDNs are random and abundant outbound links are embedded. However, extra efforts should be taken to reduce the false positives. As a matter of fact, it is also common that legitimate web sites delegate subdomains to others, and they may exhibit similar characteristics to SEO domains. We also suggest search engines to keep a close watch on keywords, especially long-tail keywords, under certain topics. As described in Section 5.4, spider pools tend to target limited topics so their campaigns could be discovered by monitoring the associated long-tail keywords.

7 Related Works

Understanding blackhat SEO. The damage caused by blackhat SEO has been known for a long time and there have been a corpus of works studying this issue. One line of such works focused on the infrastructure of blackhat SEO campaigns and how they are managed. Wang *et al.* [51] infiltrated an SEO botnet and showed it is quite effective in poisoning the trending search terms, given its small size though. John *et al.* [24] dissected an SEO campaign which hosted SEO pages on compromised servers. Also an interesting topic is about how the adversary makes use of blackhat SEO. McCoy *et al.* [34] and Levchenko *et al.* [28] revealed that blackhat SEO have been employed extensively for online store selling fake products and pharmaceutical affiliate networks. Leontiadis *et al.* [26] measured the search-redirection attacks used for drug trade and the result suggests such attacks could overwhelm legitimate sites in search results and provide more traffic than email spam to the storefront sites. To fight against these threats, search engines have been taken active actions and the studies show the countermeasures were effective to some extend but there is still long way ahead to win the battle [27, 35, 49].

Detecting blackhat SEO. Inspired by the studies of blackhat SEO campaigns, several detection approaches have been proposed based on different features revealed from known SEO attacks. John *et al.* [24] leveraged URL signatures to identify SEO pages from a dataset of URLs provided by search engines. Lu et al. [31] detected search poisoning by inspecting the redirection chains unfolded when visiting a search result. Zhang *et al.* [56] presented an approach which is capable of detecting compromised sites abused for search poisoning from seed sites. The features related to cloaking behaviors of SEO sites [50] and content spinning on SEO pages [57] were exploited for detection as well.

Wildcard DNS. Wildcard DNS is a widely used to map different hostnames to the same IP and save the administrators from maintaining many different records. The security implications, however, are not thoroughly eval-

uated yet. The only relevant research we knew so far measured the prevalence of wildcard DNS configuration and showed that it is broadly used by malicious sites [25].

Long-tail SEO Spam. Due to the intensive competition on hot keywords, many blackhat SEOers now start to target long-tail keywords. Still, the understanding is limited. So far, we only found one recent work studying this topic and showed the cloud web hosting service is abused for long-tail SEO spam [30]. Our study complements the existing works by revealing a new strategy from blackhat SEOers.

8 Conclusion

In this paper, we conducted the first comprehensive investigation on a new type blackhat SEO technique called “spider pool” which abuses wildcard DNS to tamper long-tail keywords of search engines. Based on the understanding through infiltrating a shared spider pool service, we developed a DNS prober which can identify the SEO domains with high accuracy and efficiency, together with a spider pool explorer which is able to excavate the domains used by individual spider pool through seed expansion. Our results show that spider pool has become a big threat to registrars, search engines and their users, as more than 458K SEO domains have been discovered, popular sites like `amazon.com` are abused to promote illegal messages, and long-tail keywords can be easily polluted. We think this new threat should be mitigated and call for the attention from the security community.

9 Acknowledgements

This work was supported by the Natural Science Foundation of China (grant 61472215). We thank anonymous reviewers for their insightful comments. We also owe a special debt of gratitude to Prof. Thorsen Holz, for his instructive advice on our paper. We deeply indebted to Professor Vern Paxson, his suggestion to our research direction is valuable to us. Finally, Special thanks should go to Baidu company which provide a platform and data for testing our idea.

References

- [1] ARCADIA, M. Legend of Mir Arcadia. <http://mirarcadia.com/>, 2016.
- [2] BAKII. Bakii Site Management Software (Translated). <http://www.bakii.cn/>, 2016.
- [3] BLACKHATWORLD. Wildcard Domains. Bad for website SEO? <http://www.blackhatworld.com/blackhat-seo/black-hat-seo/23514-wildcard-domains-bad-website-search-engine-optimization.html>, 2008.
- [4] BLEI, D. M. Probabilistic topic models. *Commun. ACM* 55, 4 (Apr. 2012), 77–84.
- [5] BOSONNLP. News classification; BosonNLP HTTP API 1.0 documentation (Translated). <http://docs.bosonnlp.com/classify.html>, 2016.
- [6] BOTMASTERLABS.NET. XRumer 12.0.12 Elite + Href 4.6 Professional + SocPlugin 4.0.32 + BlogsPlugin. <http://www.advancedwebranking.com/blog/how-to-identify-long-tail-keywords-for-your-seo-campaign/>, 2016.
- [7] CHINA.ORG.CN. Gestational Surrogacy Banned in China. <http://www.china.org.cn/english/2001/Jun/15215.htm>, 2011.
- [8] CHUNG, Y.-J., TOYODA, M., AND KITSUREGAWA, M. A study of link farm distribution and evolution using a time series of web snapshots. In *Proceedings of the 5th international workshop on Adversarial information retrieval on the Web* (2009), ACM, pp. 9–16.
- [9] CNNIC. .CN Domain Name : User FAQ. <http://www1.cnnic.cn/IS/CNym/cnymyfaq/>, 2016.
- [10] DEMERS, J. How to Identify Long-Tail Keywords for Your SEO Campaign. <http://www.advancedwebranking.com/blog/how-to-identify-long-tail-keywords-for-your-seo-campaign/>, 2013.
- [11] DEUTSCH, J. Confessions of a Google Spammer. <https://inbound.org/blog/confessions-of-a-google-spammer>, 2015.
- [12] DOMAINTOOLS. Domain Count Statistics for TLDs. <http://research.domaintools.com/statistics/tld-counts/>, 2016.
- [13] DOMCOMP. Domain Name Price and Availability. <https://www.domcomp.com>, 2016.
- [14] DUNN, R. Why Google Dislikes Zombie Sub-Domains. <http://www.thesempost.com/google-dislikes-zombie-sub-domains/>, 2014.
- [15] ENGE, E. The Private Blog Network Purge - Are You at Risk? <https://searchenginewatch.com/sew/how-to/2374165/the-private-blog-network-purge-are-you-at-risk>, 2014.
- [16] ENGE, E. Private Blog Networks. <http://nichesiteproject.com/private-blog-networks/>, 2015.
- [17] ENGE, E., SPENCER, S., FISHKIN, R., AND STRICCHIOLA, J. *The art of SEO*. ” O’Reilly Media, Inc.”, 2012.
- [18] FISHKIN, R. Indexation for SEO: Real Numbers in 5 Easy Steps. <https://moz.com/blog/indexation-for-seo-real-numbers-in-5-easy-steps>, 2010.
- [19] GALLAGHER, S. Many new top-level domains have become Internet’s “bad neighborhoods” [Updated]. <http://arstechnica.com/security/2015/09/many-new-top-level-domains-have-become-internets-bad-neighborhoods/>, 2015.
- [20] GOOGLE. Search Engine Optimization Starter Guide. <http://static.googleusercontent.com/media/www.google.com/en/webmasters/docs/search-engine-optimization-starter-guide.pdf>, 2008.
- [21] HALVORSON, T., DER, M. F., FOSTER, I. D., SAVAGE, S., SAUL, L. K., AND VOELKER, G. M. From .academy to .zone: An analysis of the new tld land rush. In *Internet Measurement Conference (IMC)* (2015), ACM, pp. 381–394.
- [22] HXZHANQUN. Newest spider pool templates (translated). <http://www.hxzhanzqun.com/forum-42-1.html>, 2016.
- [23] ICANN. Centralized Zone Data Service (CZDS). <https://newgtlds.icann.org/en/program-status/czds>, 2015.
- [24] JOHN, J. P., YU, F., XIE, Y., KRISHNAMURTHY, A., AND ABADI, M. deseо: Combating search-result poisoning. In *USENIX security symposium* (2011).

- [25] KALAFUT, A., GUPTA, M., RATTADILOK, P., AND PATEL, P. Surveying dns wildcard usage among the good, the bad, and the ugly. In *Security and Privacy in Communication Networks*. Springer, 2010, pp. 448–465.
- [26] LEONTIADIS, N., MOORE, T., AND CHRISTIN, N. Measuring and analyzing search-redirection attacks in the illicit online prescription drug trade. In *USENIX Security Symposium* (2011).
- [27] LEONTIADIS, N., MOORE, T., AND CHRISTIN, N. A nearly four-year longitudinal study of search-engine poisoning. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS ’14, ACM, pp. 930–941.
- [28] LEVCHENKO, K., PITSLIDIS, A., CHACHRA, N., ENRIGHT, B., FÉLEGYHÁZI, M., GRIER, C., HALVORSON, T., KANICH, C., KREIBICH, C., LIU, H., MCCOY, D., WEAVER, N., PAXSON, V., VOELKER, G. M., AND SAVAGE, S. Click trajectories: End-to-end analysis of the spam value chain. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP ’11, IEEE Computer Society, pp. 431–446.
- [29] LI, Z., ZHANG, K., XIE, Y., YU, F., AND WANG, X. Knowing your enemy: Understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS ’12, ACM, pp. 674–686.
- [30] LIAO, X., LIU, C., MCCOY, D., SHI, E., HAO, S., AND BEYAH, R. A. Characterizing long-tail SEO spam on cloud web hosting services. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016* (2016), pp. 321–332.
- [31] LU, L., PERDISCI, R., AND LEE, W. Surf: detecting and measuring search poisoning. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 467–476.
- [32] LURIE, I. SEO 101: Defining the long tail. <https://www.portent.com/blog/seo/long-tail-seo-101-defined.htm>, 2010.
- [33] MACDONALD, M. Negative SEO vs. MattCutts.com. <http://webmarketingschool.com/matt-cutts-negative-seo/>, 2013.
- [34] MCCOY, D., PITSLIDIS, A., JORDAN, G., WEAVER, N., KREIBICH, C., KREBS, B., VOELKER, G. M., SAVAGE, S., AND LEVCHENKO, K. Pharmaleaks: Understanding the business of online pharmaceutical affiliate programs. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), USENIX Association, pp. 1–1.
- [35] MOORE, T., LEONTIADIS, N., AND CHRISTIN, N. Fashion crimes: Trending-term exploitation on the web. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS ’11, ACM, pp. 455–466.
- [36] MOZILLA. Public Suffix List. <https://publicsuffix.org/>, 2016.
- [37] NIU, Y., WANG, Y.-M., CHEN, H., MA, M., AND HSU, F. A quantitative study of forum spamming using context-based analysis. Tech. Rep. MSR-TR-2006-173, Microsoft Research, December 2006.
- [38] PARK, E. Rise of .pw URLs in Spam Messages. <http://www.symantec.com/connect/blogs/rise-pw-urls-spam-messages>, 2013.
- [39] SAPE. System to attract customers (translated). <http://www.sape.ru/>, 2016.
- [40] SCHWARTZ, B. Google Penalizes Another Link Network: SAPE Links. <https://www.seroundtable.com/google-sape-link-network-16465.html>, 2013.
- [41] SECURITY, F. DNSDB. <https://www.dnsdb.info/>, 2016.
- [42] SEOMOZ. Google Algorithm Change History. <https://moz.com/google-algorithm-change>, 2016.
- [43] SHIN, Y., GUPTA, M., AND MYERS, S. The Nuts and Bolts of a Forum Spam Automator. In *Proceedings of the 4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (Mar. 2011).
- [44] SISSON, D. Google SEO Secrets. <http://www.umid.info/system/files/Google+SEO+Secrets.pdf>, 2003.
- [45] SOLUTIONS, G. new gTLD Statistics by Top-Level Domains. <https://ntldstats.com/tld>, 2016.
- [46] STATS, I. L. Google Search Statistics. <http://www.internetlivestats.com/google-search-statistics/>, 2016.
- [47] VERISIGN. Zone Files For Top-Level Domains (TLDs). https://www.verisign.com/en_US/channel-resources/domain-registry-products/zone-file/index.xhtml, 2016.
- [48] VIEWDNS. Download ccTLD domain name lists and zone files. <http://viewdns.info/data/>, 2016.
- [49] WANG, D. Y., DER, M., KARAMI, M., SAUL, L., MCCOY, D., SAVAGE, S., AND VOELKER, G. M. Search+seizure: The effectiveness of interventions on seo campaigns. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 359–372.
- [50] WANG, D. Y., SAVAGE, S., AND VOELKER, G. M. Cloak and dagger: dynamics of web search cloaking. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 477–490.
- [51] WANG, D. Y., SAVAGE, S., AND VOELKER, G. M. Juice: A longitudinal study of an SEO botnet. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013* (2013).
- [52] WARRIORFORUM. The #1 Internet Marketing Forum & Marketplace. <http://www.warriorforum.com/>, 2016.
- [53] WU, B., AND DAVISON, B. D. Identifying link farm spam pages. In *Special interest tracks and posters of the 14th international conference on World Wide Web* (2005), ACM, pp. 820–829.
- [54] ZARRAS, A., PAPADOGIANNAKIS, A., IOANNIDIS, S., AND HOLZ, T. Revealing the Relationship Network Behind Link Spam. In *13th Annual Conference on Privacy, Security and Trust (PST)* (July 2015).
- [55] ZECKMAN, A. Organic Search Accounts for Up to 64% of Website Traffic [STUDY]. <https://searchenginewatch.com/sew/study/2355020/organic-search-accounts-for-up-to-64-of-website-traffic-study>, 2014.
- [56] ZHANG, J., YANG, C., XU, Z., AND GU, G. Poisonamplifier: a guided approach of discovering compromised websites through reversing search poisoning attacks. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012, pp. 230–253.
- [57] ZHANG, Q., WANG, D. Y., AND VOELKER, G. M. Dspin: Detecting automatically spun content on the web. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014* (2014).

Appendix

A Seed Expansion through Google Search

We can start from our testing site, identify the first layer of parent sites S_1 leading to it, and recursively identify sites at upper layers $S_n(n > 1)$, till there are no more new sites discovered. We seek the help from Google Search for this task and develop a C++ program which directs Firefox to automatically search on Google. In essence, it keeps a queue of domains to be searched, pops out the first URL and queries Google, extracts all search result URLs in top 10 pages, and saves the domains unvisited into the queue. The process ends when the queue is empty.

Though the spider pool sites could be detected, the performance is poor and the process is hard to converge. We find there are two main reasons. First of all, it takes quite some time for Google to respond to our web request, and CAPTCHA has to be solved occasionally. Second, the returned search results usually include sites not in spider pool, *e.g.*, sites providing domain registration information and fake search engines which copy search results from other search engines¹³. Such irrelevant sites have to be excluded, otherwise the iterations will be inaccurate and may not even converge, but using straightforward filtering metrics like domain ranking and URL patterns are not effective through our testing.

B Classifications of SEO domains

We identified three types of methods of linking customer's content to SEO pages. They are listed below:

1. **Through iframe.** Instead of pointing to customer's site, the SEO page directly encloses the customer's page in an HTML iframe. It leverages JavaScript API `document.write` to inject the iframe, which occupies the whole screen when the user lands on the site. It could accumulate more traffic towards customer's site when visitors happen to land on SEO page.
2. **Through hyperlink.** The hyperlink pointing to customer's site or containing customer's message is displayed in the SEO page. This type is also used in SSP we infiltrated.
3. **Through redirection.** For this type, a visit to SEO page using certain user-agent string (*e.g.*, Opera browser) is immediately redirected to customer's site through HTTP 302 redirect. In fact, it uses JavaScript code to assign customer's URL to `window.top.location.href` and HTML meta refresh tag to redirect visitor's browser. Likewise,

¹³The purpose of setting up such fake search engines is not yet clear. We suspect they are used for click fraud [29].

customer's content can be directly pushed to visitor's browser.

Table 9: QQ Number (anonymized) identified for message promotion.

NO.	QQ Number	Topic	#URL
1	53*95*1	Sex	156,306
2	34*06*58*	Drugs	261,135
3	37*40*42*	Sex	225,413
4	41*88*10*	Drugs	186,370
5	51*89*17*	Drugs, Sex	224,310
6	51*45*63*	Drugs, Sex	155,459
7	63*11*95*	Drugs, Sex	150,176
8	71*40*27*	Sex	225,655
9	76*26*01*	Sex	222,864
10	77*72*10*	Drugs, Sex	224,476
11	79*43*07*	Sex	225,000
12	10*36*03*3	Sex	5,986
13	10*33*91*1	Drugs, Sex	23,944
14	10*38*34*5	Drugs, Sex	17,267
15	11*07*07*2	Drugs, Sex	19,405
16	11*35*39*2	Sex	19,883
17	11*58*92*2	Sex	16,393
18	14*14*38*6	Gambling, Sex	6,906
19	15*53*65*8	SEO	5,339
20	19*82*12*3	Gambling, Sex	6,644
21	24*50*13*5	Sex	24,949
22	24*92*74*4	Gambling	24,736
23	25*17*02*5	Gambling, Sex	6,495
Total	-	-	2,435,111

C Topics of Customer Sites

The 930 customer sites are classified under 7 categories below:

1. **Sales and Services.** We find industrial equipments and products, like elevator and seamless pipe, are sold on the customers' sites. Services of gray areas are also provided, like private detective and empty invoices.
2. **Gambling.** It includes sites for online casino and sports betting.
3. **Surrogacy.** The sites provide channels for infertile parents to find women willing to carry pregnancy.
4. **News.** The sites serve as news portals about local events.
5. **Sex.** Some of the sites host adult content like porn video and photos, while others list contacts of affiliated prostitutes or their agents.
6. **Games.** Online games developed by less known companies are provided. In addition, some sites provide information about the unauthorized servers for big-brand games, like The Legend of Mir [1], which can be connected to play the same game for free or less fees.
7. **Hospitals and Drugs.** We find the sites in this category introduce hospitals which do not have valid licenses. Besides, illegal drugs, like hallucinogen, are sold in some sites.

Table 10: Percentage of spider pool domains shown in search result of long-tail keywords.

Keyword	Type	Google	Baidu
深圳福田外围商务模特报价	Sex	no search result	80%
怎么联系深圳罗湖外围模特微信	Sex	10%	no search result
怎么联系深圳罗湖兼职模特	Sex	no search result	100%
深圳龙岗高端外围大概多少钱	Sex	10%	90%
深圳福田找高端外围女多少钱	Sex	20%	100%
深圳哪里有平面模特外围资源	Sex	no search result	100%
深圳兼职女外围经济人电话	Sex	70%	100%
深圳罗湖美空高端商务模特经济人微信号	Sex	no keyword	100%
深圳福田兼职女在哪里有	Sex	no search result	100%
深圳福田哪里找美女兼职微信	Sex	no search result	100%
深圳找白领兼职价格	Sex	no keyword	100%
深圳南山小姐一夜多少钱	Sex	100%	100%
深圳龙岗哪里能找到兼职模特	Sex	10%	90%
安眠镇定药哪里买什么价钱	Medicine	no keyword	25%
日本兴奋剂怎么在网上买	Medicine	no keyword	17.5%
男性催情药怎么买价格多少	Medicine	no keyword	20%
去哪买喷雾型迷幻药	Medicine	no keyword	20%
金阳县哪里买真的拍肩迷药	Medicine	no keyword	37.5%
龙岩市安定片怎么购买	Medicine	no keyword	30%
铜川哪里有迷情水卖的	Medicine	70%	22.5%
故城县三挫仑哪里买到	Medicine	no keyword	45%
湘乡市网上哪买迷倒药	Medicine	no keyword	30%
湛江迷幻药水哪出售	Medicine	no keyword	25%
海南省三亚市哪里有汽枪买	Firearms sales	37.5%	10%
云南省楚雄市哪里有汽枪买	Firearms sales	32.5%	15%
哪里有麻醉枪买	Firearms sales	42.5%	no keyword
仿真手枪哪里买得到	Firearms sales	37.5%	no keyword
芜湖哪里有汽枪买	Firearms sales	52.5%	no keyword
平管双猎枪	Firearms sales	30%	no keyword
大连仿真枪团购	Firearms sales	17.5%	12.5%
打鸟枪消声器货到付款	Firearms sales	20%	2.5%
舒兰汽枪买卖	Firearms sales	50%	no keyword
出售77手枪	Firearms sales	40%	no keyword
陕西代办本科毕业证	Fake certificate	37.5%	no keyword
金华市办理真实研究生本科毕业证	Fake certificate	32.5%	no keyword
牡丹江办理英语四、六级证书	Fake certificate	70%	no keyword
晋中办理英语四六级证书	Fake certificate	62.5%	no keyword
徐州办理日语二级证书	Fake certificate	80%	no keyword
茂名办自考本科文凭	Fake certificate	42.5%	no keyword
大同代开发票	Fake certificate	27.5%	17.5%
岳阳开发票公司	Fake certificate	87.5%	32.5%
哪里有制作假本科文凭的	Fake certificate	35%	12.5%
办张假本科文凭多少钱	Fake certificate	40%	12.5%

D Classifications on Customer Messages

We analyze customer messages, extract QQ numbers from customer messages, and sort them by the number of associated URLs. The result is shown in Table 9.

E Impact on Search Engines

We sampled 43 long-tail keywords extracted from the SEO pages crawled from the 21 spider pools and search them in Google and Baidu. The result is shown in Table 10. We count the percentage of search results showing spider pool domains. If no search results are returned, we write “no search result”. If no spider domains are presented in search results, we write “no keyword”.

A Comprehensive Measurement Study of Domain Generating Malware

Daniel Plohmann
Fraunhofer FKIE

Khaled Yakdan
University of Bonn

Michael Klatt
DomainTools

Johannes Bader

Elmar Gerhards-Padilla
Fraunhofer FKIE

Abstract

Recent years have seen extensive adoption of domain generation algorithms (DGA) by modern botnets. The main goal is to generate a large number of domain names and then use a small subset for actual C&C communication. This makes DGAs very compelling for botmasters to harden the infrastructure of their botnets and make it resilient to blacklisting and attacks such as takedown efforts. While early DGAs were used as a backup communication mechanism, several new botnets use them as their primary communication method, making it extremely important to study DGAs in detail.

In this paper, we perform a comprehensive measurement study of the DGA landscape by analyzing 43 DGA-based malware families and variants. We also present a taxonomy for DGAs and use it to characterize and compare the properties of the studied families. By reimplementing the algorithms, we pre-compute all possible domains they generate, covering the majority of known and active DGAs. Then, we study the registration status of over 18 million DGA domains and show that corresponding malware families and related campaigns can be reliably identified by pre-computing future DGA domains. We also give insights into botmasters' strategies regarding domain registration and identify several pitfalls in previous takedown efforts of DGA-based botnets. We will share the dataset for future research and will also provide a web service to check domains for potential DGA identity.

1 Introduction

Botnets are networks of malware-affected machines (*bots*) that are remotely controlled by an adversary (*botmaster*) through a command and control (C&C) communication channel. Botnets have become the primary means for cyber-criminals to carry out their malicious activities, such as launching denial-of-service attacks,

sending spam, and stealing personal data. Recent studies have shown that some botnets consist of more than a million bots [40], illustrating the magnitude of their threat.

Law enforcement and security researchers often try to disrupt active botnets by performing takedown attempts. The main target of these attacks is the C&C communication infrastructure of the botnet. A prominent example of these attacks is *sinkholing*, where all bots are redirected to an attacker-controlled machine called a *sinkhole*. In consequence, the bots will be prevented from communicating with the original C&C servers. As a response to these efforts, botmasters have started inventing new techniques to protect the infrastructure of their botnets. An important approach that has gained wide popularity in recent years is the use of domain generation algorithms.

A domain generation algorithm (DGA) is used to dynamically generate a large number of seemingly random domain names and then selecting a small subset of these domains for C&C communication. The generated domains are computed based on a given *seed*, which can consist of numeric constants, the current date/time, or even Twitter trends. The seed serves as a shared secret between botmasters and the bots to compute shared rendezvous points. By constantly changing the used domains, detection approaches that rely on static domain blacklists are rendered ineffective. Moreover, by dynamically generating domain names, botmasters do not have to include hard-coded domain names in their malware binaries, complicating the extraction of this information. Also, making the generated domains dependent on time lessens the value of domains extracted from dynamic malware analysis systems since different domains will be observed at different time points. Another advantage of using short-lived domains that are registered shortly before they become valid is evading domain reputation services.

The use of DGAs creates a highly asymmetric situation between attackers (botmasters) and defenders (security researchers and law enforcement). Botmasters need

access to a single domain to control or migrate their bots while defenders need to control all of the domains to ensure a successful takedown. With more than 1000 top-level domains (TLDs) [9] to choose from, it is easy for an attacker to create a global spread of responsibility for domains, forcing the defenders into additional coordination and cooperation efforts. For example, the DGA of the infamous Conficker botnet (version C) generated 50,000 domain names per day, which spread out over 113 TLDs. This required global cooperative efforts of 30 different organizations including ICANN [2] to contain the threat.

In this work we perform a comprehensive measurement study of the DGA landscape by analyzing 43 DGA-based malware families and variants. Our analysis is based on reverse-engineering the DGAs of these families. We propose a taxonomy to characterize the main aspects of DGAs and use it to describe and compare the studied DGAs. We furthermore reimplemented all of these DGAs and computed all their possible outputs based on a set of 253 seeds used in previous and ongoing malicious campaigns. We then used this set to identify DGA-generated domains in a set of 9 billion WHOIS records collected over the last 14 years. We analyze the registration status of these domains and their ownership changes, which often indicates transitioning from a malicious into a sinkholed domain. This enables us to profile the registration behaviour of both botmasters and sinkhole operators and investigate in detail the lifetime of DGA domains.

To the best of our knowledge, our work reflects the first systematic study of the DGA landscape as employed by modern botnets. Security researchers have previously examined DGAs and proposed approaches to detect and cluster DGA-based malware [13, 18, 43, 54]. Our study is instead based on an in-depth analysis of the DGAs in a bottom-up manner, by reimplementing the algorithms and enumerating the complete set of domains they generate, enabling us to have a ground truth about DGA-generated domains with no false positives.

In summary, we make the following contributions:

- We propose a taxonomy for DGAs to characterize and compare their properties.
- We analyze the DGAs of 43 malware family and variants. Using 253 identified seeds, we enumerate all possible domains generated by those algorithms, covering the majority of known and active DGAs.
- We study the registration status of 18 million DGA-generated domains covering a period of 8 years and show that corresponding malware families and related campaigns can be reliably identified by pre-computing future DGA domains.
- We analyze the strategies of both botmasters and sinkholders with regard to domain registration and

identify several pitfalls in previous takedown efforts of DGA-based botnets.

- We share the dataset for future research and provide a web service called DGArchive [38] to check whether a queried domain originates from a DGA.

2 A DGA Taxonomy

In this section, we propose a taxonomy for DGAs to characterize their properties and enable a comparison. To model the entire spectrum of different properties, we propose two features designed to capture the different aspects of a domain generation algorithm. Both features are then combined in a single taxonomy that classifies DGAs into classes.

2.1 Seed Source

The seed serves as a shared secret required for the calculation of generated domains, also referred to by the term Algorithmically-Generated Domains (AGD) [54]. It is the aggregated set of parameters required for the execution of a domain generation algorithm. Typical parameters include numerical constants (e.g., length of domains or seeds for pseudo random number generators) or strings (e.g., the alphabet or the set of possible TLDs).

Two properties of seeding have superior significance to characterize a DGA (cp. Barabosch et al.[16]):

Time dependence means that the DGA incorporates a time source (e.g. the system time of the compromised host or the date field in a HTTP response) for calculation of AGDs. In consequence, generated domains will have a validity period only during which these domains are queried by the compromised system.

Determinism addresses the observability and availability of parameters. For the majority of known DGAs, all parameters required for DGA execution are known to a degree that all possible domains can be calculated. Two DGAs use temporal non-determinism to disallow arbitrary prediction of future AGDs by using unpredictable but publicly accessible data for seeding. The malware family Bedep [44] makes use of foreign exchange reference rates published daily by the European Central Bank while a later variant of Torpig [50] used Twitter trends for seeding. In both cases, this only leads to attackers and defenders having to compete for the registration of domains in each active time window once the unpredictable data used for seeding becomes available. However, it does not prevent historic analysis, as the seeding data and thus generated domains can still be collected over time. Another kind of non-determinism has been observed by Symantec [45]. Their analysis of Jiripbot revealed that the malware exfiltrates a set of system properties including MAC address and hard drive volume ID

to make it available to the attacker to be used in a DGA seed. In this case, the system information is considered non-deterministic to the attacker prior to compromise and also never publicly observable.

Time-dependence and determinism allow the following four combinations: time-independent and deterministic (TID), time-dependent and deterministic (TDD), time-dependent and non-deterministic (TDN), time-independent and non-deterministic (TIN). In our dataset, we have only observed DGAs using the first three classes of seeding properties.

2.2 Generation Schemes

Apart from the characteristics of seeding, 4 different generation schemes emerged during our analysis.

Arithmetic-based DGAs calculate a sequence of values that either have a direct ASCII representation usable for a domain name or designate an offset in one or more hardcoded arrays, constituting the alphabet of the DGA. They are the most common type of DGA.

Hash-based DGAs use the hexdigest representation of a hash to produce an AGD. We identified DGAs using MD5 and SHA256 to generate domains.

Wordlist-based DGAs will concatenate a sequence of words from one or more wordlists, resulting in less randomly appealing and thus more camouflaging domains. These wordlists are either directly embedded in the malware binary or obtained from a publicly accessible source.

Permutation-based DGAs derive all possible AGDs through permutation of an initial domain name.

We abbreviate the generation scheme with the respective starting letter: A, H, W, or P. As part of any of the above-mentioned generation schemes, some DGAs leverage pseudo-random number generators (PRNGs) to generate domains. These range from implementing own PRNGs to the use of well-known techniques such as linear congruential generators (LCGs) [36].

2.3 DGA Types

As DGA type, we consider the combination of seeding properties and generation scheme, denominated by the combined abbreviations, e.g. “TID-A” for a time-independent, deterministic DGA using a arithmetic-based domain generation scheme. Of 16 possible combinations, we have only observed 6 types being used by the 43 DGAs in our dataset: TDD-A (20), TID-A (16), TDD-W (3), TDD-H (2), TDN-A (1), TID-P (1).

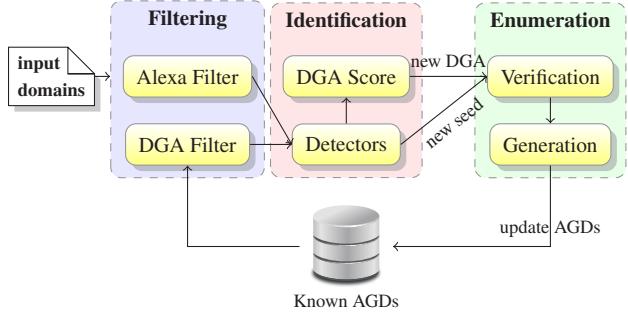


Figure 1: DGA collection approach.

3 DGA-Malware Dataset

In this section, we describe how we identified and collected malware samples that employ a DGA. Then, we describe our efforts to reverse-engineer the algorithms, reimplementing and evaluating their implementation.

3.1 Identifying DGA-based Malware

The first step of our study was to collect a representative set of DGA-based malware families. To this end, we developed a system to automatically identify potentially new DGAs by analyzing a set of domain names generated by a given malware sample. This helps us to minimize the set of malware samples we need to manually reverse-engineer, and thus focus our efforts on samples that are likely to implement new DGAs. A high-level overview of the system is presented in Figure 1. First, we filter out known AGDs and benign domains. Second, we identify domains that are generated by a previously known DGA but with a new seed, and domains potentially generated by new DGAs. Third, if a new DGA or a new seed is identified, we manually reverse-engineer the corresponding sample, extract the seed and the algorithm, and compute the set of domains it generates. In the following, we discuss these steps in detail.

Filtering. We first filter out known benign and popular domains by comparing them against the first 10,000 entries of the Alexa list of top-ranked domain names [10]. Second, we filter out known AGDs by comparing the input domains against our current collection of enumerated AGDs. A good starting point for collecting DGA-based malware samples are malware analysis reports and blogs. Using these as sources and based on our experience, we identified an initial set of 22 families using DGAs. In 9 of these cases, we already found a reimplementation of the DGA that we only had to verify. This enables us to compute an initial set of domains to use in the DGA filter.

Identification. In this step we identify samples that implement a previously known DGA but use new seeds, and potentially new DGAs. To that end, we use a set of *detectors* implemented as regular expressions to quickly decide for a given domain if it matches the expected output of one of these known DGAs. The regular expressions catch major characteristics such as minimum and maximum length of the generated part (L_{min} and L_{max}), DGA alphabet Σ , and the set of known TLDs. If a majority of input domains is matched by the same pattern, we mark the sample as using a known DGA with new seed.

If the detectors produce an inconsistent or insufficient result, we compute a *collective DGA score* for the input domains to measure the likelihood of these domains being generated by a DGA. This score is based on features such as n-gram frequency, entropy, and length analysis. If available, we further increase the score in case of an *NX domain* result. This approach relies on previous work on DGAs [13, 18, 34, 43, 54]. If the score exceeds a threshold derived through prior experiments and manual verification, we mark the sample as a candidate for using a new unknown DGA.

DGA enumeration. Whenever the system identifies a potentially new DGA, we manually verified this by reverse-engineering the corresponding sample. If present, we then extracted the domain generation logic from the binary, reimplemented it, identified the used seed, and finally computed all domains generated by the algorithm. Moreover, by analyzing the algorithm we updated the set of detectors to identify future samples that use the same algorithm with different seeds.

As input to our system, we used a special sandbox feed that was kindly provided by the Shadowserver Foundation, consisting of timestamp, malware sample hash, DNS query, and DNS response. This feed contains a mix of both newly observed malware samples and older samples undergoing re-processing, dating back to at least 2009. This re-processing increases the likelihood to encounter unknown seeds and families that may no longer be active. Taking data of 3 months starting in May 2015 from this feed, we examined a total of 1,235,443 sandbox runs, performing a total of 15,660,256 DNS queries towards 959,607 unique domain names. Based on the data of the Shadowserver feed, our system enabled us to identify, analyze, and re-implement another 21 DGAs and the majority of seeds listed in Table 3.

3.2 Reimplementing DGAs

To ensure our reimplementation is correct, we compared its output against the domain queries issued by the respective malware sample when executed in a sandbox. For each family, the reverse-engineering and reimplementation of the corresponding DGA took around one

day. After analyzing the algorithm, we extracted the DGA seed from the binary. In many cases, we fully automated the process of seed extraction, which enabled us to easily extract seeds from new binaries of the same family. After unpacking the malware binaries, we rarely encountered further protection layers of the original unpacked code, which allowed considerable analysis speed. Nymaim and Suppobox samples are heavily obfuscated and employ family-specific code obfuscations. To handle these cases, we analyzed the obfuscation techniques and implemented custom deobfuscators to automatically deobfuscate these samples. Pykspa 2 was an exceptionally difficult case. Instead of reverse-engineering the seed derivation function, it was much easier to instrument it in order to generate all possible seed values that we later used in the reimplementation of the DGA.

The manual analysis of samples previously identified by the system helped us to eliminate several samples that would otherwise be false positives. For example, during the evaluations of the sandbox DNS feed, we came across many samples whose DNS queries appeared like typical AGDs but were in fact hardcoded domains as verified through our reverse-engineering. Furthermore, we only consider DGAs in which the AGDs are actually used as valid, potential C&C endpoints and not as distractions like in the following cases. For instance, we found an early variant of Sality [1] prepending dynamically generated third-level parts to hardcoded domains, which is the earliest hint to DGA-like behavior we were able to identify. Another example is the Zeus-derivate Citadel, which produces unused decoy AGDs when having detected a virtualized environment [21].

We are aware that more malware families than listed in this paper are potentially using DGAs. However, we believe that the given data set provides a sufficient basis to draw meaningful conclusions on a majority of effects caused by domain generation algorithms.

4 Insights into the DGA Landscape

In this section, we present a comprehensive overview of the 43 DGA implementations and their 253 seeds that we collected. We use our taxonomy to characterize and compare the studied DGAs. More specifically, we compare domain structure, validity periods, and generation schemes. We also study how random the generated domains are and the priority of the DGA as C&C communication mechanism. Table 1 presents an overview of the different DGA features. The table includes different variants of some families with different DGAs. This is the case for Gameover Zeus, Murofet, Pushdo, and Pykspa.

Name	Reference	DGA Type	$C2_{prio}$	Valid	$ D_{cycle} $	L_{min}	L_{max}	TLDs	$ \Sigma $	H_{rel}
Bamital	[37]	TDD-H (MD5)	1/1	1d	104	32	32	4	16	1.000
Banjori	[14]	TID-A	2/2*	∞	2,196-15,373	11	26	1	26	0.948
Bedep	[44]	TDDN-A	1/1	7d	22-28	12	18	1	36	0.944
Conficker	[25]	TDD-A	2/2	1d	250-50,000	4	11	123	26	1.000
Corebot	[14]	TDD-A (NR LCG)	2/2	1d	40	12	23	1	34	1.000
CryptoLocker	[3]	TDD-A	1/1	1d	1,000	12	15	7	25	1.000
DirCrypt	[14]	TID-A (PM LCG)	1/1	∞	30	8	20	1	26	0.999
Dyre	[5]	TDD-H (SHA256)	3/3	1d	1,000	34	34	8	36	0.805
Feodo	-	TID-A (NR LCG)	1/1	∞	64	16	18	1	26	0.993
Fobber	[47]	TID-A (own LCG)	1/1	∞	1,000	10	17	2	26	1.000
Gameover DGA	[14]	TDD-A (MD5)	1/1	1d	1,000/10,000	20	28	4	36	0.983
Gameover P2P	[11]	TDD-A (MD5)	2/2	1-7d	1,000	11	32	6	26	1.000
Geodo	[30]	TDD-A	1/1	900s	time-based	16	16	1	25	1.000
Gootkit	[48]	TDD-A (PM LCG)	1/1	12h	1	16	16	1	26	0.997
Gozi	[4]	TDD-W (NR LCG)	1/1	1-3mo	5-80	12	24	12	26	0.883
Hesperbot	[27]	TID-A	2/2	∞	50-64	8	24	1	26	0.997
Kraken	[41]	TID-A	1/1	∞	300	6	11	4	26	0.998
Matsnu	[49]	TDD-W	2/2*	3d	3	12	24	1	27	0.895
Mewsei	[14]	TDD-A (MS LCG)	1/1	16d	64	8	15	1	23	0.939
Murofet 1	[14]	TDD-A (MD5)	1/1	1d	1,020	8	16	5	26	0.965
Murofet 2	[14]	TDD-A (MD5)	1/1	1-7d	1,000	32	47	6	36	0.994
Necurs	[14]	TDD-A	2/2	4d	2,048	7	21	43	25	1.000
Nymaim	[15]	TDD-A (Xorshift)	2/2*	1d	30	6	11	8	26	1.000
Pushdo	[6]	TDD-A (MD5)	2/2	60d	30	8	12	2	26	0.962
Pushdo TID	-	TID-A (NR LCG)	1/1	∞	6,000	10	10	5	26	1.000
Pykspa 1	[14]	TDD-A	1/1	2d	5,000	6	15	6	26	0.963
Pykspa 2	[14]	TDD-A (own LCG)	1/1	1-20d	1,000	6	12	4	26	0.998
QakBot	-	TDD-A (MT, CRC32)	1/1	8-11d	5,000	8	25	5	26	1.000
Ramdo	[29]	TID-A	1/1	∞	1,000	16	16	1	13	1.000
Ramnit	[46]	TID-A (PM LCG)	2/2	∞	1,000	8	19	1	25	1.000
Ranbyus	[14]	TDD-A	1/1	28-31d	40	14	14	8	25	1.000
Redyms	[32]	TID-A	1/1	∞	34	9	15	1	27	0.990
Rovnix	-	TID-A (MS LCG)	2/3	∞	10,000	18	18	5	34	0.999
Shifu	[24]	TID-A (own LCG)	2/2	∞	777	7	7	1	25	1.000
Simda	[14]	TID-A	1/1	∞	1,000	5	11	4	26	0.965
Suppobox	[22]	TDD-W	1/1	12h	168	8	26	1	26	0.889
Szribi	[52]	TDD-A	2/2	1-3d	4	8	8	1	15	0.949
Tempedreve	[7]	TID-A (own LCG)	1/1	∞	204	7	11	4	26	0.996
TinyBanker	[14]	TID-A	2/2*	∞	100-4,000	12	12	15	25	0.987
Torpig	[50]	TDD-A	1/1	1d	3	7	9	3	30	0.937
UrlZone	[14]	TID-A	2/2*	∞	2,000	9	15	2	32	1.000
Virut	[20]	TDD-A (Delpi LCG)	2/2	1d	100*100	6	6	1	26	0.984
VolatileCedar	[17]	TID-P	2/2	∞	170	14	14	1	9	0.959

Table 1: Overview of studied DGA implementation characteristics. *Type* according to our taxonomy. $C2_{prio}$ lists the priority of DGA among all C&C rendezvous mechanisms found (with * indicating hardcoded domains being redundant or used as part of seed). *Valid* describes the duration and D_{cycle} the number of domains generated per period. $L_{min,max}$ denotes extrema of domain length, *TLDs* is a combined value over all seeds of this DGA. Σ is the alphabet used in AGDs, H_{rel} normalized entropy.

4.1 Domain Structure

Alphabet. We first study the alphabet, denoted by Σ , used by the DGAs to generate domains. The alphabets contain between 9 and 36 characters. While some DGAs use intentionally short alphabets, the majority of DGAs with small alphabets seems to result from flawed implementations. An example of the first case is VolatileCedar, which generates domains by permuting the second-level part of a hard-coded domain (`dotnetexplorer.net`). Ramdo’s DGA is an example of a buggy implementation. Only letters with odd indexes are chosen when building the domain name (i.e., a, c, e, . . .), resulting in an alphabet with only 13 letters. Hash-based DGAs have alphabets of 16 characters consisting of digits and letters from a to f.

Many DGAs use a hard-coded array of characters from which is chosen by an index computed iteratively. A

common bug in these DGAs are *off-by-one* errors, where one or more characters are never chosen. For example, Geodo omits the last character of its alphabet, thus generated domains never contain a z. Other DGAs with this bug include CryptoLocker, Necurs, Ramnit, Ranbyus, Shifu, TinyBanker, and Torpig. They all miss the last character in their alphabet. This type of error is worse for DGAs that use multiple separate arrays to choose characters from. For example, Mewsei uses two arrays, one for vowels and one for consonants (intentionally or not, missing the letter j). An off-by-one bug results in one character in each array to be missed, reducing the effective alphabet size by two. Rovnix and Corebot use two separate arrays of letters and digits and suffer from the same malfunction, causing their AGDs to never contain a z or 9. We found this bug in 11 of the studied DGAs.

We also observed *truncation* errors, where the alphabet is truncated into a smaller array that is used for computing the domains. For example, Szribi truncates its 26 intended hard-coded letters to only 15. The same error occurs in Urlzone, reducing the possible 35 characters to only 32. Torpig suffers from both bugs: Torpig’s randomness is flawed in a way that only 30 out of 34 possible (36 minus 2 resulting from the off-by-one error) are reachable. The largest alphabet of 36 symbols is used Dyre, Gameover DGA, and Murofet 2.

AGD length. The length of generated domains ranges from 4 to 47 with median minimum length 9 and median maximum length 16. 14 DGAs produce AGDs that all have identical length. In three cases (Banjori, Simda, and Torpig), we found multiple length values across seeds but all AGDs of one seed have the same length.

Domains levels. Only three families (Corebot, Kraken, and Mewsei) generate third-level domains, using one or more Dynamic DNS providers. All remaining DGAs only generate a second-level domain that is then concatenated with a top-level part. Note that we consider constructs as `co.uk` or `com.tw` as top-level part since they are managed by a single registry. Conficker and Necurs make extensive use of TLDs with 123 and 43 TLDs respectively. These cases strikingly illustrate the need of global cooperation to successfully sinkhole some botnets. This also holds for botnets with DGAs that use a smaller set of TLDs. Table 2 shows the most popular TLDs used by the studied DGAs and identified seeds. The 7 most common TLDs are the same in both listings and they only differ in their order. `com` and `net` are the two top TLDs. We observe a trend to use popular TLDs with no regional reference, as `com` and `net` together make up about 45% of all registered domains [8]. We assume that attackers want their generated domains to blend in well with benign traffic.

TLD	per DGAs		per Seeds	
	Occurrences	TLD	Occurrences	TLD
com	28	com	178	
net	21	net	82	
org	16	ru	56	
info	15	biz	40	
biz	13	in	40	
ru	10	info	32	
in	6	org	29	
cc	5	pw	26	
su	5	su	21	
eu	4	cc	18	

Table 2: Popularity of TLDs, both on for DGAs and seeds.

4.2 Domain Validity Periods

More than half of the studied DGAs (24/43) are time-dependent, meaning that the generated domains are only

valid for a certain period of time. Most of these DGAs (21/24) generate domains with disjunct validity periods. That is, only a single set of domains is valid at each point in time. The three exceptions are 1) Matsnu, which generates 3 domains which are each valid for 3 consecutive days, meaning that there are 9 potential C&C domains at a time; 2) Pushdo, which will generate domains relative to a given date, starting 45 days in the past and up to 15 days in the future. With 30 domains per day, this gives 900 domains valid at a time; and 3) Suppobox, producing one AGD per 512 seconds, which is then valid for the next 85 periods, totalling to 12 hours, 5 minutes, 20 seconds per AGD.

11 time-dependent DGAs generate domains that are valid for one day. Other families increase their domain validity by performing certain calculations on the date. For example, both Gameover P2P and Murofet 2 round the day of month down to the next lowest value of 1, 7, 14, 21, or 28, resulting in validity periods of variable length (1-7 days). Qakbot employs a similar scheme but uses different values for rounding (1, 11, and 21). The domains generated by Szribi’s DGA [52] are valid for 1-3 days. This variable period length appears to be a bug since the DGA tries to round the day to the nearest third day (i.e., Julian days).

Both versions of Pykspa have unique characteristics with regard to the validity of generated domains. Pykspa 1 generates a list of 5000 domains every two days, and the validity of each domain depends on its position in this list. The first 20 domains are generated at random while the remaining 4,980 are chosen from two static sets of domains, alternating between these sets every two days. This is caused by the DGA raising the initial 32-bit seed to power of 2 for each new computed domain, thus drastically reducing the randomness of computed domains. In order to increase resilience to detection, Pykspa 2 generates *fake* domains with shorter validity periods than the real ones. While 200 real domains are generated that are valid for 20 days, the DGA generates 800 fake domains, each of which is only valid for one day.

Geodo has a unique generation strategy with regard to validity periods among all time-dependant DGAs. It queries the current date from the response of an HTTP request to a Microsoft website. Then, it will consequently generate one AGD every 900 seconds starting from a hard-coded start date until this current date is reached.

4.3 Generation Schemes

Arithmetic-based DGAs are by far the most common generation scheme (37/43). Among these, 26 DGAs directly compute the ASCII codes of the characters to be used in the domain, while 11 DGAs compute an in-

dex that is used to select characters from hard-coded arrays representing the used alphabet. Three DGAs are wordlist-based: Matsnu, Suppobox, and Gozi. Matsnu and Suppobox embed the list of words in their corresponding malware binary. On the other hand, Gozi extracts its wordlist from a publicly available text file, which is very unlikely to be changed in the future. For example, it uses the United States Declaration of Independence to generate domains such as `amongpeaceknownlife.com`. Matsnu and Gozi randomly combine words until a certain length is reached, while Suppobox only combines 2 of its 384 included words and adds `.net` as TLD.

The two families Bamital and Dyre are hash-based and use the hexdigest output of hashing functions MD5 and SHA256 over date and domain index as input parameters. VolatileCedar is the only permutation-based family, which can produce 170 possible permutations of the initial domain. It is worth mentioning that Banjori, TinyBanker, Urlzone and VolatileCedar use a domain mutation scheme where either a seed or a previously generated domain is used as input for the calculation of the subsequent domain.

4.4 Domain Randomness

Given that various DGAs make use of PRNGs in their domain generation schemes, we analyzed the randomness of generated domains. To this end, for each DGA, we compute the global string by concatenating all generated parts of the domains. Then we calculate the Shannon entropy of this string. This gives a global indicator for the randomness of the algorithm. Finally, we compute the relative entropy, denoted by H_{rel} , by dividing the calculated value with the maximum entropy. This enables us to compare the relative entropy of different DGAs. For simplicity, we use the term entropy to refer to H_{rel} .

We make the following observations. All wordlist-DGAs have a significantly lower entropy ($H_{rel} < 0.9$), which is expected since they combine complete words whose characters are not uniformly distributed. However, the lowest entropy is observed in Dyre, a hash-based DGA. While the SHA256 algorithm produces uniformly distributed characters, Dyre prepends an extra character in the range `[a-z]` to the calculated hexdigest, thus destroying the uniform distribution since the hexdigest does not contain any letters in the range `[g-z]`.

The low entropy ($H_{rel} < 0.99$) observed in other DGAs is a result of specific implementation choices. We discuss this in the following.

1) **Multiple sub-alphabets.** By splitting the alphabet into multiple distinct lists and randomly drawing characters from these lists for different positions in the com-

puted domain name, some DGAs cause an imbalance on the overall distribution of characters. This is the case for Bedep, Gameover DGA, Mewsei, Pushdo, Simda, Torpig, and Virut. Redyms uses a hyphen in every AGD.

2) **Imperfect PRNG.** We observed several causes of imperfect PRNGs: first, some DGAs (Pykspa 1 and Szribi) use their own self-designed PRNGs, which have imperfect randomness. Second, other DGAs (Murofet 1 and TinyBanker) impose certain conditions on the output of the used PRNGs, and thus cause an imbalance in the distribution of derived characters.

3) **Partial modifications.** Some DGAs compute the next domain based on an initial domain name. This transformation may impact the DGA entropy if it does not introduce enough modifications in the initial domain seed. An example of this category is Banjori, which only modifies the first four positions of a given hard-coded seed domain.

It is worth mentioning that several DGAs use well-known PRNG algorithms. Notably, almost a third of the studied DGAs (14/43) use a linear congruential generator (LCG) [36] defined by the recurrence relation $X_{n+1} = (aX_n + c) \bmod m$. By matching the used parameters a , c , and m against those of common LCGs, we found that 10 DGAs use known LCG constants: 4 DGAs use the implementation described in the Numerical Recipes book [39] (NR LCG), 3 DGAs use the *minimal standard* implementation proposed by Park and Miller [36] (PM LCG), 2 DGAs use the Microsoft Visual C library (MS LCG), and one DGA uses the Delphi LCG. The remaining 4 LCGs use self-chosen constants. Other uses of well-known PRNGs we identified are Mersenne Twister [33] and Xorshift [31]. Five DGAs use MD5 for initializing their own PRNG and one uses the checksum CRC32.

4.5 Command & Control Priority

More than half of the studied botnets (23/43) use their DGA as the only C&C rendezvous mechanism. Moreover, although 5 other families (identified by a star in Table 1) first try hard-coded domains before turning to their DGAs, the DGAs can also be considered as the primary communication mechanism. Banjori, TinyBanker, and UrlZone first try a single hard-coded domain that is later used in computing the initial seed of the corresponding DGA. Matsnu contains multiple prioritized hard-coded domains. However, the same domains are also generated by its DGA at some point in time. Nymaim contains a *legacy* primary hard-coded domain that is long mitigated and probably disregarded by the botmasters, meaning that it mainly relies on its DGA instead.

Other families use their DGAs as a backup mechanism when their primary communication method (usu-

ally hard-coded domains) fails. For example, Rovnix tries to connect to one of its hard-coded domains. If this fails, it resorts to its DGA before finally trying a connection via Invisible Internet Project (I2P). The remaining families use their DGAs as a last resort to reach the C&C server.

This clearly shows the prevalence of DGAs in modern botnets. For this reason, we believe that DGAs should be no longer perceived mainly as a backup mechanism but instead as a primary C&C concept.

5 DGA Domain Usage

In this section, we present the results of our analysis on how DGAs are actually used in practice. We base our analysis on WHOIS data for the last 14 years, and give insights into the family activity periods, registration status of DGA domains, and botmaster registration strategies. We also analyze the mitigation response time.

5.1 WHOIS Dataset

We based our analysis on the DomainTools WHOIS dataset, which contains over 9 billion WHOIS records collected over the last 14 years [19]. Based on our reimplementations and the available seeds, we computed all possible domains generated by the studied 43 DGAs. We provided the computed domains to DomainTools, and they kindly provided us with WHOIS records for DGA domains they found in their dataset. The data provided by DomainTools for this study was compiled on the 22nd September of 2015. This enabled us to identify 303,165 DGA-related WHOIS records for 115,387 domains.

A special care had to be given to time-dependant DGAs, which use the date information for computing their domains. To ensure a good coverage and minimize the set of DGA domains we miss, we computed all domains that cover a time period of one year before any publicly known starting date until 31st December 2015 (three months after the last WHOIS record in the DomainTools dataset). This end date provides around 3 months of *lookahead*, detecting domains that are registered for some time before being actually used.

These WHOIS records contain the following fields

- Date of the WHOIS record
- Date of updates to the WHOIS record
- Dates for domain creation and expiration
- Registrar name
- Registrant name and e-mail address
- Nameservers registered in WHOIS

This enabled us to infer the following information:

- 1) *Domain first registration*. This is based on the date of the WHOIS record and date of domain creation.

2) *Start of botnet activity*. The first record for any domain implicitly indicates that the domain is registered and helps to estimate when the corresponding malware family started its operation.

3) *Changes in domain ownership*. The date of updates to a WHOIS record often corresponds to a change in responsibility for a domain. This can be verified by accompanying changes to the registrar, registrant, and registered nameserver fields. This allows us to derive when a potential C&C domain has been mitigated (e.g., sinkholed), by observing changes of the WHOIS record from a non-sinkhole operator to a sinkhole operator. We identify 29 different organizations running sinkholes, but we will not disclose further details about the features used to detect sinkhole organizations to protect their operations.

4) *Parked domains*. The registrar, registrant, and registered nameserver fields convey information that we use to identify parked domains in order to investigate how common AGDs are held for the purpose of picking up accidental traffic or reselling. In total, we identify 36 services offering domain parking or domain reselling.

Our dataset covers all but five of the studied families: Conficker and Virut generate an exceedingly large number of domains, and were therefore not included in the dataset provided to us. Corebot, Kraken, and Mewsei use DDNS-based DGAs, and are thus not available in the original DomainTools WHOIS dataset, which only contains primary domain names but no subdomains. It is noteworthy that non-deterministic seeding as used by Bedep does not prevent retroactive analysis, as we are still able to generate all domains by collecting the used exchange rates over time.

5.2 Family Activity Periods

In this section, we analyze the period of time in which the studied botnets were active and domains were registered for malicious purposes. To this end, we first identify the date of the first related record in our dataset, denoted by T_{first} . In many cases, this is the first observed registration overall. In cases where the data indicates collisions with benign domains (cp. Section 5.4), we analyze records from the first sinkhole event backwards. Then, we identify the time when the domains were either taken down or they were registered last, denoted by T_{last} . The detailed activity periods for the studied botnets are shown in Table 3. For families we did not have data for, their respective T_{first} is estimated from public sources as mentioned before.

In case of Bamital, CryptoLocker, and Gameover P2P, T_{last} is the date of their takedown, marked by a †. For Conficker, T_{last} identifies the date on which the last known Conficker variant deleted itself from compromised systems [26]. A special case is Rovnix, for which

Name	T_{first}	T_{last}	$ S $	$ D_{gen} $	$ D_{uniq} $	$ R_{all} $	$\frac{ R_{all} }{ D_{uniq} }$	$ R_P $	$ R_M $	$ R_S $
Kraken	2007-07*	-	1	300	300	-	-	-	-	-
Torpig	2008-01	2011-06-22	2	17,610	17,610	139	0.79%	2	1	57
Szribi	2008-11	2011-06-22	1	4,396	2,949	54	1.83%	0	8	40
Conficker	2008-11*	2009-05-03†	3	129,807,750	125,118,625	-	-	-	-	-
Pushdo TID	2009-07	2012-04-06	1	6,000	6,000	245	4.08%	0	0	0
Pykspa 1	2009-10	2012-09-09	1	32,920	22,764	455	2.00%	12	0	49
Gozi	2010-01	2015-09-18	9	21,890	16,963	305	1.80%	48	4	143
Murofet 1	2010-08	2011-09-08	2	4,063,680	4,063,680	3,172	0.08%	0	50	369
Bamital	2010-11	2013-02-06†	1	197,600	197,600	8,340	4.22%	0	150	30 (7,891)
Nymaim	2011-06	2015-09-16	3	277,112	65,040	656	1.01%	70	17	388
Simda	2011-06	2014-11-06	12	13,000	11,528	379	3.29%	66	9	44
Ramnit	2011-06	2015-02-07	18	18,000	18,000	939	5.22%	0	126	372
Virut	2011-08*	-	1	16,140,000	15,355,008	-	-	-	-	-
Murofet 2	2011-09	2011-12-20	1	262,000	262,000	559	0.21%	0	4	261
Gameover P2P	2011-09	2014-05-28†	1	262,000	262,000	74,755	28.53%	0	23	391 (72,713)
Feodo	2012-02	2012-10-06	3	192	192	110	57.29%	0	1	9
Gootkit	2012-06	2013-11-08	1	2,190	730	198	27.12%	0	0	4
Redyms	2012-12	2014-02-10	1	34	34	11	32.35%	0	2	2
Necurs	2013-01	2015-06-12	6	3,551,232	3,551,232	295	0.01%	10	0	158
CryptoLocker	2013-01	2014-05-30†	1	1,108,000	1,108,000	3,820	0.34%	0	341	240 (2,899)
Suppobox	2013-02	2015-09-20	3	545,169	98,304	11,338	11.53%	8,434	19	792
Banjori	2013-03	2013-09-10	30	434,556	421,390	683	0.16%	0	3	33
Pushdo	2013-03	2015-08-05	4	124,080	124,021	453	0.37%	3	0	54
Pykspa 2	2013-04	2013-10-01	2	775,400	775,342	1,927	0.25%	757	5	101
VolatileCedar	2013-04	2015-03-30	1	170	170	13	7.65%	0	0	7
DirCrypt	2013-07	2014-06-15	14	420	420	86	20.48%	0	13	21
Hesperbot	2013-07	2015-01-07	3	178	178	15	8.43%	0	1	10
Ramdo	2013-10	2014-05-03	3	3,000	3,000	47	1.57%	0	5	23
UrlZone	2013-11	2015-09-20	6	12,006	10,009	127	1.27%	0	24	34
QakBot	2013-12	2015-09-20	1	385,000	385,000	1,088	0.28%	0	61	35
Matsnu	2014-01	2015-09-20	2	3,375	3,346	610	18.23%	244	33	61
Dyre	2014-06	2015-08-19	1	592,000	592,000	850	0.14%	0	1	273
Gameover DGA	2014-07	2014-11-21	2	6,182,000	6,182,000	1,081	0.02%	0	14	549
TinyBanker	2014-08	2015-09-21	90	84,291	81,930	1,733	2.12%	0	272	326
Geodo	2014-10	2014-11-16	2	90,240	90,232	107	0.12%	0	0	39
Tempedreve	2014-10	2015-04-19	1	204	204	20	9.80%	0	0	13
Mewsei	2014-10*	-	1	1,984	1,984	-	-	-	-	-
Fobber	2014-10	2015-07-01	2	2,000	2,000	13	0.65%	0	2	4
Ranbyus	2015-01	2015-08-10	7	105,840	64,400	98	0.15%	0	0	36
Rovnix	2015-01*	-	1	10,000	10,000	1	0.01%	0	0	1
Bedep	2015-02	2015-09-20	4	3,906	3,806	654	17.18%	0	10	201
Corebot	2015-06*	-	2	18,160	18,160	-	-	-	-	-
Shifu	2015-07	2015-09-10	2	1,554	1,554	11	0.71%	0	0	8
Aggregated	-	-	253	165,161,439	159,712,234	115,387	0.63%	9,646	1,199	5,177 (83,503)

Table 3: Overview of DGA usage characteristics. $|S|$: seeds known for this DGA. D : domains generated until 31.12.2015, R : registered domains ($|R_P|$: prior to T_{first} ; $|R_M|$: turned into a sinkhole; $|R_S|$: directly registered as sinkhole; in brackets: related to botnet takedowns). Registration percentage based on D_{uniq^*} , thus considering only AGDs where registration data was available.

we only identified a single registration by a sinkhole operator, and thus decided to leave T_{last} open.

After identifying the starting date for each family (T_{first}), we removed the AGDs we computed for earlier points in time. This reduces the computed set to 165,161,439 AGDs in total, 159,712,234 of them being unique across all DGAs. Additionally, we remove the computed domains for the families not available in the DomainTools dataset, leaving us with a set of 18,446,125 unique AGDs. We denote this set by D_{uniq^*} and use it for all following examinations.

Having a closer look at T_{first} shows that although the concept has been first used in 2007, more than half of the analyzed DGAs (25/43) were introduced 2013 and later. It seems like malware authors waited for early adopters to gather experience with running DGA-based botnets before trying it out themselves. Cases with short activ-

ity times indicate that some authors seemingly also only experimented with DGAs. For example, while early versions of Geodo used hard-coded IP addresses for C&C communication, one version with a DGA was used for only about 6 weeks. After that, the botmasters returned back to the old communication method.

Murofet 2 remained active for three months only, before its widespread successor Gameover P2P, which used a very similar DGA, appeared. Gameover DGA, another successor that appeared shortly after the takedown of Gameover P2P, was active for only about 5 months. Although binary diffing implies that it relies on the same source code as Gameover P2P, it is uncertain whether it was operated by same original author Slavik [42] or if it was only used as a distracting maneuver.

We use T_{last} to identify active families as of this writing. We consider a family to be active if its DGA gener-

ated a domain that was used within the last month of the time period covered by the WHOIS dataset. Based on that, we identify 10 families: Gozi, Nymaim, Suppobox, UrIZone, QakBot, Matsnu, Dyre, TinyBunker, Bedep, and Shifu. This serves as a lower bound for estimating active families since some do not use DGA as their primary C&C rendezvous mechanism.

5.3 Domain Registration Status

Out of the 18,446,125 unique generated AGDs, 115,079 were actually registered (0.62%). 72.37% of these registered AGDs correspond to sinkhole operators, while the remaining 27.63% are non-takedown domains. We divide the set of registered AGDs into four categories:

- 1) **Pre-registered domains (30.25%).** Domains that have been registered before T_{first} . These domains were usually registered long before the corresponding botnet appeared, and are very likely benign.
- 2) **Mitigated domains (3.76%).** Domains that were originally held by a non-sinkhole registrant but then changed to a sinkhole operator, usually indicating that the domain has been mitigated (Section 5.6).
- 3) **Pure sinkhole domains (16.24%).** Domains registered by a sinkhole operator from the start.
- 4) **Remaining domains (49.75%).** This set contains domains that were registered between T_{first} and T_{last} but we were not able to reliably identify them as sinkholes, or otherwise tell whether they are benign or malicious (mostly due to the use of a WHOIS privacy service). However, the number of benign domains (at least for non-wordlist DGAs) is highly likely to be a minority, given that few DGAs have pre-registered domains and the common randomness and domain length produced by most DGAs.

5.4 Domain Collisions

In order to evaluate if AGDs are a good feature to reliably identify the corresponding botnet, we analyze collisions between domains generated by different DGAs. Out of the 43 DGAs, the two aggressive DGAs Virut and Conficker have collisions among each other and with Necurs, Nymaim, and Pykspa 2. More specifically, Virut has 1791 collisions with Pykspa 2, 1316 with Conficker, and 179 with Nymaim. Conficker has 11 collisions with Pykspa 2, 4 with Necurs, and 1 with Nymaim. All of the colliding domains are 5 to 6 characters long. Apart from that, only Nymaim and Pykspa 2 have a single collision since they both generate the AGD `wttttf.net`.

Next, we analyze collisions between DGA-generated domains and benign domains. To this end, we use the Alexa list [10] and compare it to the set of domains generated by each DGA. Again here, only a small num-

ber of collisions were found. Virut has 2507 collisions with Alexa top million domains, which corresponds to 0.016% of its unique AGDs. For all other DGAs, we only found 27 collisions with the Alexa list: 24 collisions with wordlist-based DGAs (Suppobox: 21, Matsnu: 2, Gozi: 1), and three collisions with Pykspa 2 domains that are 6 character long.

We then investigated the collisions between DGA-generated domains and domains that were already registered before the corresponding botnet appeared (before T_{first}). As one would expect, the highest number of collisions were observed with wordlist-based DGAs: 74.39% of Suppobox AGDs and 40.0% of Matsnu AGDs collide with already existing domains. In spite of being a wordlist DGA, Gozi has a low number of collisions with already registered domains. This is because the DGA truncates, with a probability of 33%, a chosen word before appending it to the computed AGD. This creates many domains that contain a *broken* word, making accidental registration far less likely. For the remaining families, 856 of 928 (92.24%) pre-registered AGDs have length 5 to 6, and everything longer is accidentally a word (e.g. `veterans.kz`) or otherwise very pronounceable (e.g. `kankanana.com`, `kandilmed.com`).

These results clearly show that DGAs not based on wordlists and generating domain names longer than 6 characters (which is the case for 34/43 DGAs) serve as a reliable source to detect the corresponding botnet family.

5.5 Domain Registration Lookahead

A special property of AGDs from time-dependent DGAs is, that they are only valid during certain periods of time. In this section, we give insights into the domain registration *lookahead* for time-dependant DGAs, i.e., the amount of time from registering the domain until the point in time where it becomes valid (produced by the DGA at that time). Here, we distinguish between sinkhole and non-sinkhole registrations (potentially by botmasters).

Figure 2 provides an overview of our results for the 22 time-dependant DGAs in our dataset. For each family, the results are summarized using a boxplot showing the registration times of DGA domains relative to the start of the corresponding validity periods (day zero). That is, a registration event at $x = -1$ means that the corresponding domain was registered one day before it became valid. For better visualization, we choose a symmetric mixed linear (for the first 10 days) and logarithmic (everything beyond) scale. This allows best to display data with a finer resolution around the majority of validity periods but also allows to include events up to a thousand days away. For better orientation, the red bars indicate the whole validity period of AGDs per family. In this Fig-

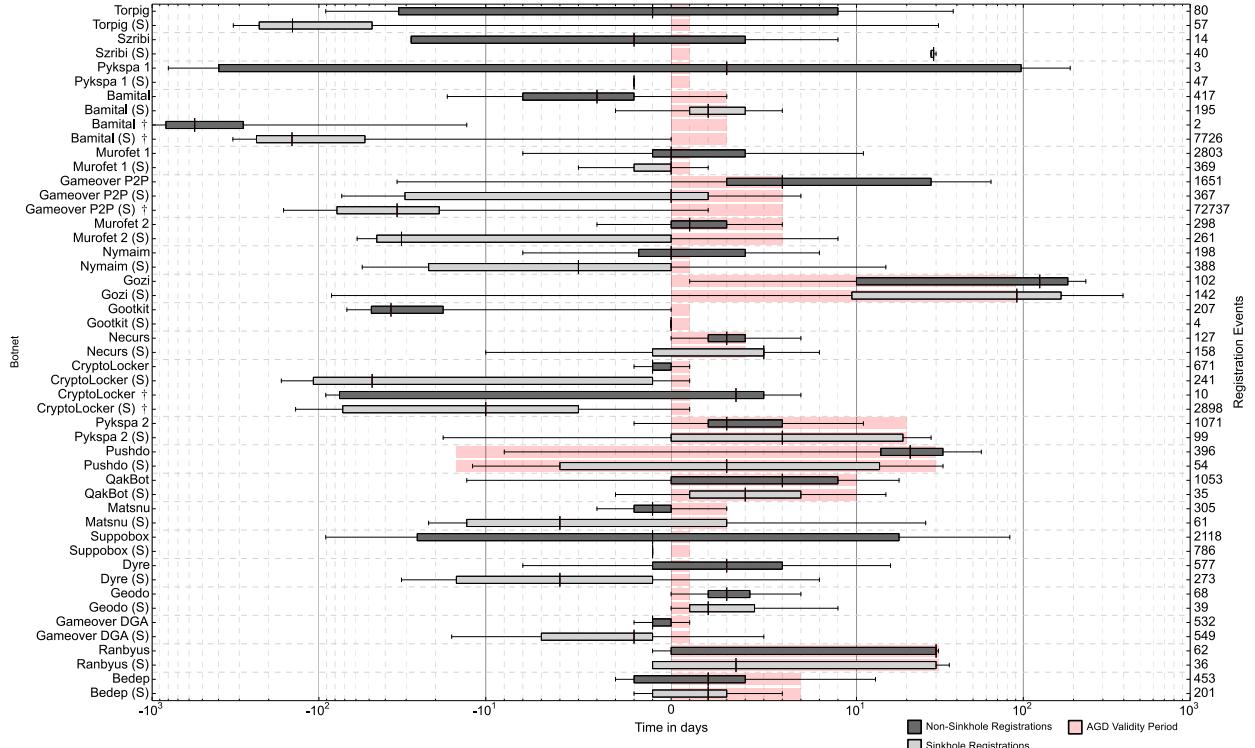


Figure 2: Lookahead of domain registrations in time-dependent DGAs, divided into identifiable sinkholes and remaining domains. The data in this boxplot is applied relative to the start the respective AGD’s validity period (shown in light red for better orientation). Bamital, GameoverP2P, CryptoLocker data is further divided into pre and post takedown (indicated by †).

ure, we distinguish between four possible categories for each family: 1) none sinkhole registrations; 2) sinkhole (marked by S) registrations; 3) pre-takedown registrations; and 4) post-takedown registrations (marked by †). Note that the registration numbers may vary from those in Table 3. This is a result of plotting data after T_{first} with respect to validity start instead of the registration dates.

The main observation is that for 14 of 22 DGAs, sinkhole operators registered the domains earlier than non-sinkholders. Comparing the medians of these cases reveals that in 7 cases sinkhole registrations happen between 1 and 10 days earlier, while in the remaining 7 cases they happen between 18 and 143 days earlier. The 143 days are for Torpig, where the majority of sinkhole domains were registered 5 months in advance with close validity periods, which seems to be a takedown attempt. The second highest difference (48 days) was for CryptoLocker. For 3/22 DGAs, sinkhole and non-sinkhole registrations happened at the same time. In the remaining 5/22 DGAs, non-sinkhole registrations occurred between 2 and 37 days before sinkhole registrations.

In the case of Gootkit, a single sinkhole operator registered DGA domains on the same day when they became valid, indicating a reactive response to these domains be-

ing used by the malware but not being taken yet, potentially blocking the botmaster from registering them.

Sinkhole operators usually do not fear that their domains will be taken away by other entities, which explains the general trend of sinkholders registering domains before botmasters. Moreover, this enables them to evaluate incoming traffic for these domains even before they become valid. On the other hand, botmasters have to assume that their domains will eventually be mitigated. As a result, they register their domains shortly before or inside the validity period in order to ensure the availability of these domains when the bots are expected to try to contact them. This means that constant monitoring of AGDs with validity periods around the takedown date is a necessary condition for success.

For the three families with takedowns, sinkhole registrations happened at least weeks and often months prior to validity of the AGDs. This ensures that botmasters cannot quickly regain control of their botnet by registering further AGDs. While the takedown of Gameover P2P’s AGDs was seemingly complete, AGDs for both Bamital and CryptoLocker were registered by non-sinkholders after the takedown. In case of CryptoLocker, these domains were registered shortly before

their validity started, indicating that not all domains remain blocked through the same central authority.

We observed an interesting pattern in the case of Bamital. On January 1st, 2011, AGDs for January 4th, 2012, January 3rd, 2013, and January 4th, 2014 domains were registered by the same registrant. We believe that this was done by the operators of Bamital in an attempt to timely secure *insurance domains* to be used as backup in case of a later takedown, speculating that these domains do not get discovered. We identified similar strategies by the botmasters of Nymaim and Murofet. In case of Nymaim, on December 6th, 2012 and December 9th, 2012, two domains becoming valid in 1 and 2 years respectively were registered. For Murofet, on March 12th, 2011 and the 3 following days, 4 domains becoming valid in 1 year, 5 valid in 2 years, 3 valid in 3 years, and 3 valid in 4 years have been registered. All of them use different, apparently fake identities but the same registrar. This shows that potential registrations of AGDs should be checked considerably far ahead during the preparation of a takedown campaign.

The number for sinkhole registrations of Pykspa 1 and Suppobox is impressively small. For these families, most AGDs are registered with the same lookahead by single sinkhole operators over longer periods of time. Registering only a small fraction of available domains indicates botnet monitoring operations by sinkholders. On the other hand, we observed many different entities involved on non-sinkhole registrations of Suppobox domains. Given the comparatively high collision rate with benign domains (Section 5.4), we believe that many of these domains belong to benign registrants. This shows that the Suppobox DGA blends in very well with legitimate domain owners, allowing the botmasters to hide their C&Cs among benign domains.

We observed a few peculiarities with regard to domain registrations. Registrations for Gozi tend to be late for both sinkholders and others. We found no coherent explanation for this. Another oddity are the sinkhole registrations for Szribi. While 8 registrations happened in 2008 when the botnet was active, another 32 registrations happened in 2015, more than 4 years after the botnet disappeared. Moreover, all of these 32 domains were registered one month after they became invalid. We believe that the respective sinkholing organization used a wrong DGA reimplementation to calculate the AGDs.

5.6 Mitigation Response Time

By mitigation we mean a change of domain ownership from a non-sinkhole to a sinkhole operator. In this section, we analyze the time offset for these events in order to measure the effectiveness of these operations. Here, we only consider DGAs where 10 or more mitigations

were identified that are not related to takedowns. Table 4 summarizes the results of this examination. In most cases, the first mitigations against new DGAs or seeds are carried out within a week.

One would expect that after initial identification of a DGA or a new seed for a DGA, all further mitigations would have a much lower response time as the potentially generated domains should be known. We observe relatively short median reaction times \tilde{m} for Murofet 1, Suppobox, Gameover DGA, and TinyBanker. These are also the DGAs where most AGDs were mitigated within their respective validity periods. On the other hand, the countermeasures for Bamital and CryptoLocker have been very ineffective as they mostly targeted AGDs that were no longer valid.

Response times for all other DGAs increase after the first mitigation. Interestingly, multiple follow-up mitigations for these DGAs were carried out on the same day. This pattern corresponds to a common practice by defenders. After identifying initial malicious domains, they look at other domains that point to the same C&C server by performing a reverse IP lookup. This identifies additional AGDs which may have been registered even before the initially identified AGD. This was observed for Nymaim, Ramnit, and UrlZone.

5.7 DGAs and Domain Parking

Next, we examine how many AGDs are registered towards domain parking services. For identification of such domains, we used a procedure similar to Vissers et al. [51]. In total, we found 6,458 AGDs that contained a registration pointing to a parking service at some point in time. This corresponds to the common practice that after the registration period of an original owner ends, the expired domain is transferred to domain parking or picked up by a domain reseller.

To our surprise, for 3,852 of these AGDs, the parking registration was also the first and in many cases only registration entry. In particular, first parking registration events constitute a significant portion of registrations for the following DGAs: Banjori 620 (90.78%), QakBot 595 (54.69%), Pykspa 2 883 (45.82%), Necurs 122 (41.36%), Pushdo TID 91 (37.14%), Ramnit 286 (30.46%), Murofet 1 736 (23.20%). Except for Pykspa 2, none of these DGAs has a significant number of pre-registrations $|R_P|$, which makes unintended or accidental registration extremely unlikely. It is also noteworthy, that for these DGAs, 2,917 (87.52%) domains are registered with the same domain parking service. With regard to validity periods, 464 out of 2336 (19.86%) time-dependent AGDs are registered before the end of the respective validity periods. In this special case, together with the time-independent AGDs, at least 1461 AGDs were potentially

Name	Seeds	Mitigations	Validity Period		First Response (in days)				Further Responses (in days)			
			within	after	m_{min}	\bar{m}	\tilde{m}	m_{max}	m_{min}	\bar{m}	\tilde{m}	m_{max}
Murofet 1	1	50	37	13	25	25	25.00	25	0	0	1.77	25
Bamital	1	148	16	132	6	6	6.00	6	3	49	47.37	92
Nymaim	2	16	4	12	1	16	16.00	31	0	5	31.46	212
Ramnit	16	126	-	-	2	19	34.60	153	0	35	54.28	363
CryptoLocker	1	216	25	191	9	9	9.00	9	0	8	14.69	130
Suppobox	2	19	13	6	3	117	117.50	232	0	0	25.12	194
DirCrypt	7	13	-	-	0	3	7.00	26	0	5	10.50	41
UrlZone	4	24	-	-	0	4	6.75	19	0	114	112.40	251
QakBot	1	33	15	18	0	0	0.00	0	0	21	28.41	66
Matsnu	2	33	11	22	2	2	2.50	3	2	7	9.16	72
Gameover DGA	2	14	9	5	0	1	1.50	3	0	1	54.17	161
TinyBanker	51	272	-	-	0	3	6.39	61	0	2	5.66	60
Bedep	2	10	4	6	2	5	5.00	8	1	12	13.12	28

Table 4: Mitigation Response Timings for selected DGAs. For time-dependent DGAs, *Validity Period* describes the identified mitigations for active and outdated AGDs. *First Response* is the time until the first mitigation occurred, with values for minimum, median, average, and maximum, aggregated over seeds. *Further Responses* describes the same measures for all following events.

abused to drive automatically generated traffic from compromised hosts towards a domain parking system.

5.8 Discussion: Countering DGAs

In this section, we summarize the observations made in previous sections and discuss how they influence countermeasures against DGAs.

Looking at the distribution of domain generation schemes (Section 4.3), only 3 of 43 DGAs are based on wordlists and therefore produce somewhat meaningful domain names. On the other hand, methods for the detection of domains that appear randomly generated are well-studied [13, 34, 54] and our data suggests that they remain relevant and applicable in the future.

As the study of activity periods has shown (Section 5.2), DGAs have become very relevant to malware authors, especially over the last 2 years, as 25 out of the 43 considered DGAs surfaced 2013 and later.

One of the core concepts of DGAs is the implied economic asymmetry: A single valid domain grants an attacker control while the defender needs to deny access to all potential domains. The overall low number of actually registered AGDs (Section 5.3) underlines the fact that attackers only need to make sparse use of all potential domains when operating their botnets. Additionally, since most attackers seem to register domains very shortly before their validity or use (Section 5.5), there is a high chance that they can be hit by surprise and the perceived backup utility of DGAs can be cancelled. However, having identified registration events up to 4 years upfront means future domains should be carefully checked to ensure successful takedowns.

Our data indicates that in past takedowns, domains were acquired on a large scale, imposing significant financial efforts connected to the takedown. Having do-

mains registered as consequence of a takedown allows them to be used as sinkholes in order to gather telemetry on compromised systems calling in. However, single AGDs per validity period would be enough to achieve the same monitoring effect, similar to how an attacker only needs a single domain, as long as the remaining domains are not available. Therefore, we propose to raise awareness of the relevance and innerworkings of DGAs to ICANN and make extensive use of blocking AGDs on the level of registry operators. The expected impact of blocking would actually be negligible for the majority AGDs: 34/43 of the DGAs analyzed have AGDs with minimum length 7 or more, for which we observed basically no collisions with existing domains for their entire set of AGDs (Section 5.4), meaning that these domains seem unlikely to be registered for benign purposes anyway. In case of time-dependent DGAs, this blocking could even be lifted once the validity period of the respective AGDs has passed, which could be a compromise to address wordlist-based DGAs.

Furthermore, we identified only 3,302 collisions between 5 DGAs within 159,712,234 unique DGA domains (Section 5.4). This means that a lookup database of AGDs like our data set serves as a very reliable resource to aid the identification of malware families based on contacted domains with basically no false positives. As a service to the community, we continue to collect information on DGAs and provide this data for free through DGArchive [38].

6 Related Work

Among others, the following works have addressed Domain Generation Algorithms in detail. Stone-Gross et al. [50] have performed a botnet takeover for Torpig in early 2009. Based on reverse-engineering, they took ad-

vantage of the fact that this family was using a Domain Generation Algorithm without further protection mechanisms in the C&C protocol. Their publication was one of the first descriptions detailing the concept of DGAs in academic literature. Barabosch et al.[16] have defined a taxonomy of DGA types based on the 2 features time-dependency and causality. They also explained a method for the automated localization of DGA-related code using dynamic analysis for unpacking and API tracing in combination with data flow analysis for code extraction. Mowbray et al. [34] have used collected DNS data to identify potential DGA domains examining the query source IP address and length distribution of queried domain names. With this approach they identified 19 different schemes of AGDs and list a subset of characteristics of those given in Table 1.

Several works target the detection of DGAs and other maliciously used domains based on collected network traffic. In 2010, Yadav et al. [54] have evaluated several statistical measures over character distributions and n-grams in domain names in order to detect generated domain names through anomalies. Antonakakis et al. [13] have presented Pleiades, a DNS-based system that detects clusters of potential DGA domains by monitoring unsuccessful DNS requests. As confirmed by our work, they built their system on the assumption that only a fraction of AGDs is actually registered. They were able to find 6 new DGAs that were unknown at the time by evaluating traffic from a large ISP. Another approach using DNS data named Phoenix is proposed by Schiavoni et al. [43]. They have defined a model for pronounceable domains and afterwards detect those domains deviating from this model. Their system also groups identified domains and allows tracking the activity of DGA-based botnets. Bilge et al. [18] have introduced their system Exposure. It is used to detect malicious domain names based on a selection of 15 features observable for DNS traffic, including time-based, DNS answer-based, TTL, and domain name features.

There are also works that explore capabilities of dynamic and proactive blacklisting. In 2009, Ma et al. [28] have compared the usefulness of different information sources for blacklisting. They found out that WHOIS data, especially temporal information of registrations, are very valuable in this regard. Antonakakis et al. [12] have proposed Notos, a system to automatically assign scores to domain names that can be used to automatically generate blacklists. They focus on features of domain strings and TLDs. Xu et al. [53] have mentioned the idea to pre-generate AGDs in order to enable predictive blocking. However, they did not evaluate the effectiveness of this idea, as shown by us in this paper.

Recently, Vissers et al. [51] have investigated the relationship of domain parking services and malicious do-

mains, using DNS and WHOIS data sets. Nadji et al. [35] have developed a concept for effective botnet takedowns in which they identify covering potential DGA presence as important step for effectiveness. With regard to the analysis of further innovations in C&C rendezvous mechanisms, Holz et al. [23] have investigated Fast-Flux Service Networks, while Rossow et al. [40] have performed a survey of botnets using P2P mechanisms.

7 Conclusion

In this work, we presented the first comprehensive measurement study of domain generation algorithms as used by modern botnets. Our study is based on reverse-engineering the DGAs of 43 malware families and variants. Using reimplementations of the algorithms, we generated a collection of 159,712,234 unique DGA domains. We then performed an analysis on domain registrations, utilizing historic WHOIS data provided by DomainTools. Our main findings are that our domain dataset can be used for both predictive blocking of attempted C&C accesses as well as the accurate determination of malware families and campaigns with basically no false positives. Additionally, we characterized the registration behavior of botmasters and sinkholders and examined the effectiveness of domain mitigations.

As a further contribution, we continuously collect further information on DGAs. The full domain data set which results from our work is published for free here: <https://dgarchive.caad.fkie.fraunhofer.de>. This web service called *DGArchive* offers reverse domain lookups to support malware analysis, as well as forward generation of domain lists, which can be particularly used as blocklists for network protection.

Acknowledgments

The authors would like to express eternal gratitude to the Shadowserver Foundation for continuously supporting malware research. We would also like to thank the anonymous reviewers of USENIX Security as well as Daniela Bennewitz, Arnold Sykosch, and Matthias Wübbeling for their valuable feedback.

References

- [1] W32/sality.m, February 2006. Malware description by McAfee: <http://www.mcafee.com/threat-intelligence/malware/default.aspx?id=138354>.
- [2] Conficker Working Group: Lessons Learned. Tech. rep., The Rendon Group, <http://www.confickerworkinggroup.org>, January 2011.
- [3] Ransom Cryptolocker. Tech. rep., McAfee Labs Threat Advisory, November 2014.

- [4] Tracking Rovnix, 2014. Blog post: <http://labs.bitdefender.com/2014/11/tracking-rovnix-2/>.
- [5] Chasing cybercrime: network insights of Dyre and Dridex Trojan bankers. Tech. rep., Blueliv, 2015.
- [6] Pushdo It To Me One More Time. Tech. rep., Fidelis Cybersecurity, April 2015.
- [7] Tempedreve - Botnet overview and malware analysis. Tech. rep., Anubisnetworks, 2015.
- [8] The Domain Name Industry Brief - Volume 12, Issue 3. Tech. rep., Verisign, 2015.
- [9] TLD DNSSEC Report, 2015. Statistics page published by ICANN: http://stats.research.icann.org/dns/tld_report/.
- [10] ALEXA. Top sites on the Web, 2015. Website: <http://www.alexa.com/topsites>.
- [11] ANDRIESSE, D., ROSSOW, C., STONE-GROSS, B., PLOHMANN, D., AND BOS, H. Highly resilient peer-to-peer botnets are here: An analysis of Gameover Zeus. In *Proceedings of the 8th International Conference on Malicious and Unwanted Software (MALWARE)* (2013).
- [12] ANTONAKAKIS, M., PERDISCI, R., DAGON, D., LEE, W., AND FEAMSTER, N. Building a Dynamic Reputation System for DNS. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association.
- [13] ANTONAKAKIS, M., PERDISCI, R., NADJI, Y., VASILOGLOU, N., ABU-NIMEH, S., LEE, W., AND DAGON, D. From Throw-away Traffic to Bots: Detecting the Rise of DGA-based Malware. In *Proceedings of the 21st USENIX Conference on Security Symposium* (2012).
- [14] BADER, J. Domain Generation Algorithm analyses, 2015. Blog posts on various DGAs: <http://www.johannesbader.ch/tag/dga/>.
- [15] BARABOSCH, T. Behavior-Driven Development in Malware Analysis: Can it Improve the Malware Analysis Process?, 2015. Presentation: <https://itsec.cs.uni-bonn.de/spring2015/downloads/barabosch.pdf>.
- [16] BARABOSCH, T., WICHMANN, A., LEDER, F., AND GERHARDS-PADILLA, E. Automatic Extraction of Domain Name Generation Algorithms from Current Malware. In *Proceedings of the NATO Symposium IST-111 on Information Assurance and Cyber Defence* (2012).
- [17] BAUMGARTNER, K., AND RAIU, C. Sinkholing Volatile Cedar DGA Infrastructure, 2015. Blog post: <https://securelist.com/blog/research/69421/sinkholing-volatile-cedar-dga-infrastructure/>.
- [18] BILGE, L., SEN, S., BALZAROTTI, D., KIRDA, E., AND KRUEGEL, C. Exposure: A Passive DNS Analysis Service to Detect and Report Malicious Domains. *ACM Trans. Inf. Syst. Secur.* 16, 4 (Apr. 2014).
- [19] DOMAINTOOLS. Company Profile, 2015. Website: <https://www.domaintools.com/company/>.
- [20] FALLIERE, N. W32.Virut: Using Cryptography to Prevent Domain Hijacking, 2011. Blog post: <http://www.symantec.com/connect/blogs/w32virut-using-cryptography-prevent-domain-hijacking>.
- [21] GASTESI, M., AND GEGENY, J. Citadel Updates: Anti-VM and Encryption change, June 2012. Blog post for S21sec: <http://securityblog.s21sec.com/2012/06/citadel-updates-anti-vm-and-encryption.html>.
- [22] GEFFNER, J. End-To-End Analysis of a Domain Generating Algorithm Malware Family. In *Proceedings of the 2013 Blackhat Conference* (2013).
- [23] HOLZ, T., GORECKI, C., RIECK, K., AND FREILING, F. Measuring and Detecting Fast-Flux Service Networks. In *Proceedings of the 15th Annual Network & Distributed System Security Conference (NDSS)* (2008).
- [24] KESSEM, L. Shifu: 'Masterful' New Banking Trojan Is Attacking 14 Japanese Banks, 2015. Blog post: <https://securityintelligence.com/shifu-masterful-new-banking-trojan-is-attacking-14-japanese-banks/>.
- [25] LEDER, F., AND WERNER, T. Know Your Enemy: Containing Conficker, To Tame a Malware. Tech. rep., The Honeynet Project, <http://honeynet.org>, 2009.
- [26] LEUNG, K., LIU, Y., AND KIERNAN, S. W32.Downadup.E Technical Details. Tech. rep., Symantec, 2009.
- [27] LIPOVSKY, R. Hesperbot - A new, Advanced Banking Trojan in the Wild. Tech. rep., ESET, 2013.
- [28] MA, J., SAUL, L. K., SAVAGE, S., AND VOELKER, G. M. Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2009), KDD '09, ACM, pp. 1245–1254.
- [29] MALWARE PROTECTION CENTER. MSRT April 2014 on Ramdo, 2014. Malware description by Microsoft: <http://blogs.technet.com/b/mmpc/archive/2014/04/08/msrt-april-2014-ramdo.aspx>.
- [30] MALWARE PROTECTION CENTER. Trojan:Win32/Emotet.C, 2014. Malware description by Microsoft: <https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Trojan:Win32/Emotet.C>.
- [31] MARSAGLIA, G. Xorshift RNGs. *Journal of Statistical Software* 8, 1 (2003).
- [32] MATROSOV, A. What do Win32/Redyms and TDL4 have in common, 2013. Blog post: <http://www.welivesecurity.com/2013/02/04/what-do-win32redyms-and-tdl4-have-in-common/>.
- [33] MATSUMOTO, M., AND NISHIMURA, T. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (Jan. 1998).
- [34] MOWBRAY, M., AND HAGEN, J. Finding Domain-Generation Algorithms by Looking at Length Distribution. In *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Naples, Italy* (2014).
- [35] NADJI, Y., ANTONAKAKIS, M., PERDISCI, R., DAGON, D., AND LEE, W. Beheading Hydras: Performing Effective Botnet Takedowns. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2013), CCS '13, ACM.
- [36] PARK, S. K., AND MILLER, K. W. Random Number Generators: Good Ones Are Hard to Find. *Commun. ACM* 31, 10 (Oct. 1988).
- [37] PIOTR KRYSIUK, V. T. Trojan.Bamital. Tech. rep., Symantec, 2013.
- [38] PLOHMANN, D. DGArchive. Fraunhofer FKIE: <https://dgarchive.caad.fkie.fraunhofer.de>.
- [39] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3 ed. Cambridge University Press, New York, NY, USA, 2007.

- [40] ROSSOW, C., ANDRIESSE, D., WERNER, T., STONE-GROSS, B., PLOHMANN, D., DIETRICH, C. J., AND BOS, H. SoK: P2PWNED — Modeling and Evaluating the Resilience of Peer-to-Peer Botnets. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)* (San Francisco, CA, May 2013).
- [41] ROYAL, P. On the Kraken and Bobax Botnets. Tech. rep., Damballa, April 2008.
- [42] SANDEE, M. GameOver ZeuS - Backgrounds on the Badguys and the Backends. Tech. rep., Fox IT, 2013.
- [43] SCHIAVONI, S., MAGGI, F., CAVALLARO, L., AND ZANERO, S. Phoenix: DGA-Based Botnet Tracking and Intelligence. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2014), vol. 8550 of *Lecture Notes in Computer Science*.
- [44] SCHWARZ, D. Bedep’s DGA: Trading Foreign Exchange for Malware Domains, 2015. Blog post: <https://assert.arbornetworks.com/bedeps-dga-trading-foreign-exchange-for-malware-domains/>.
- [45] SECURITY RESPONSE. Butterfly: Corporate spies out for financial gain. Tech. rep., Symantec, July 2015.
- [46] SECURITY RESPONSE. W32.Ramnit Analysis. Tech. rep., Symantec, February 2015.
- [47] SEGURA, J. Elusive HanJuan EK Drops New Tinba Version, 2015. Blog post: <https://blog.malwarebytes.org/intelligence/2015/06/elusive-hanjuan-ek-caught-in-new-malvertising-campaign/>.
- [48] SINEGUBKO, D. Runforestrun and Pseudo Random Domains, June 2012. Blog post for Unmask Parasites: <http://blog.unmaskparasites.com/2012/06/22/runforestrun-and-pseudo-random-domains/>.
- [49] SKURATOVICH, S. Matsnu. Tech. rep., Check Point Software technologies Ltd., May 2015.
- [50] STONE-GROSS, B., COVA, M., CAVALLARO, L., GILBERT, B., SZYDŁOWSKI, M., KEMMERER, R., KRUEGEL, C., AND VIGNA, G. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009).
- [51] VISSERS, T., JOOSEN, W., AND NIKIFORAKIS, N. Parking Sensors: Analyzing and Detecting Parked Domains. In *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium* (2015).
- [52] WOLF, J. Technical details of Srizbi’s domain generation algorithm, November 2008. Blog post for FireEye: <https://www.fireeye.com/blog/threat-research/2008/11/technical-details-of-srizbis-domain-generation-algorithm.html>.
- [53] XU, W., SANDERS, K., AND ZHANG, Y. We Know It Before You Do: Predicting Malicious Domains. In *Proceedings of the 24th Virus Bulletin Conference (VB2014)* (2014).
- [54] YADAV, S., REDDY, A. K. K., REDDY, A. N., AND RANJAN, S. Detecting Algorithmically Generated Malicious Domain Names. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (2010), IMC ’10.

Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing

Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly,

Ismail Khoffi, Linus Gasser, and Bryan Ford

EPFL

Abstract

While showing great promise, Bitcoin requires users to wait tens of minutes for transactions to commit, and even then, offering only probabilistic guarantees. This paper introduces ByzCoin, a novel Byzantine consensus protocol that leverages scalable collective signing to commit Bitcoin transactions irreversibly within seconds. ByzCoin achieves Byzantine consensus while preserving Bitcoin’s open membership by dynamically forming hash power-proportionate consensus groups that represent recently-successful block miners. ByzCoin employs communication trees to optimize transaction commitment and verification under normal operation while guaranteeing safety and liveness under Byzantine faults, up to a near-optimal tolerance of f faulty group members among $3f + 2$ total. ByzCoin mitigates double spending and selfish mining attacks by producing collectively signed transaction blocks within one minute of transaction submission. Tree-structured communication further reduces this latency to less than 30 seconds. Due to these optimizations, ByzCoin achieves a throughput higher than Paypal currently handles, with a confirmation latency of 15-20 seconds.

1 Introduction

Bitcoin [47] is a decentralized cryptocurrency providing an open, self-regulating alternative to classic currencies managed by central authorities such as banks. Bitcoin builds on a peer-to-peer network where users can submit transactions without intermediaries. Special nodes, called *miners*, collect transactions, solve computational puzzles (*proof-of-work*) to reach consensus, and add the transactions in form of blocks to a distributed public ledger known as the *blockchain*.

The original Bitcoin paper argues that transaction processing is secure and irreversible, as long as the largest colluding group of miners represents less than 50% of

total computing capacity and at least about one hour has elapsed. This high transaction-confirmation latency limits Bitcoin’s suitability for real-time transactions. Later work revealed additional vulnerabilities to transaction reversibility, double-spending, and strategic mining attacks [25, 31, 34, 35, 48, 3].

The key problem is that Bitcoin’s consensus algorithm provides only probabilistic consistency guarantees. Strong consistency could offer cryptocurrencies three important benefits. First, all miners instantly agree on the validity of blocks, without wasting computational power resolving inconsistencies (*forks*). Second, clients need not wait for extended periods to be certain that a submitted transaction is committed; as soon as it appears in the blockchain, the transaction can be considered confirmed. Third, strong consistency provides *forward security*: as soon as a block has been appended to the blockchain, it stays there forever. Although increasing the consistency of cryptocurrencies has been suggested before [17, 19, 43, 52, 56], existing proposals give up Bitcoin’s decentralization, and/or introduce new and non-intuitive security assumptions, and/or lack experimental evidence of performance and scalability.

This work introduces ByzCoin, a Bitcoin-like cryptocurrency enhanced with strong consistency, based on the principles of the well-studied Practical Byzantine Fault Tolerance (PBFT) [14] algorithm. ByzCoin addresses four key challenges in bringing PBFT’s strong consistency to cryptocurrencies: (1) open membership, (2) scalability to hundreds of replicas, (3) proof-of-work block conflicts, and (4) transaction commitment rate.

PBFT was not designed for scalability to large consensus groups: deployments and experiments often employ the minimum of four replicas [38], and generally have not explored scalability levels beyond 7 [14] or 16 replicas [16, 32, 1]. ByzCoin builds PBFT atop CoSi [54], a collective signing protocol that efficiently aggregates hundreds or thousands of signatures. Collective signing reduces both the costs of PBFT rounds and the costs

for “light” clients to verify transaction commitment. Although CoSi is not a consensus protocol, ByzCoin implements Byzantine consensus using CoSi signing rounds to make PBFT’s *prepare* and *commit* phases scalable.

PBFT normally assumes a well-defined, closed group of replicas, conflicting with Bitcoin’s open membership and use of proof-of-work to resist Sybil attacks [23]. ByzCoin addresses this conflict by forming consensus groups dynamically from *windows* of recently mined blocks, giving recent miners *shares* or voting power proportional to their recent commitment of hash power. Lastly, to reduce transaction processing latency we adopt the idea from Bitcoin-NG [24] to decouple transaction verification from block mining.

Experiments with a prototype implementation of ByzCoin show that a consensus group formed from approximately the past 24 hours of successful miners (144 miners) can reach consensus in less than 20 seconds, on blocks of Bitcoin’s current maximum size (1MB). A larger consensus group formed from one week of successful miners (1008) reached consensus on an 8MB block in 90 seconds, showing that the system scales both with the number of participants and with the block size. For the 144-participant consensus group, with a block size of 32MB, the system handles 974 transactions per second (TPS) with a 68-second confirmation latency. These experiments suggest that ByzCoin can handle loads higher than PayPal and comparable with Visa.

ByzCoin is still a proof-of-concept with several limitations. First, ByzCoin does not improve on Bitcoin’s proof-of-work mechanism; finding a suitable replacement [4, 28, 37, 58] is an important but orthogonal area for future work. Like many BFT protocols in practice [15, 32], ByzCoin is vulnerable to slowdown or temporary DoS attacks that Byzantine nodes can trigger. Although a malicious leader cannot violate or permanently block consensus, he might temporarily exclude minority sets ($< \frac{1}{3}$) of victims from the consensus process, depriving them of rewards, and/or attempt to censor transactions. ByzCoin guarantees security only against attackers who consistently control less than a third (not 50%) of consensus group shares – though Bitcoin has analogous weaknesses accounting for selfish mining [25].

In this paper we make the following key contributions:

- We use collective signing [54] to scale BFT protocols to large consensus groups and enable clients to verify operation commitments efficiently.
- We present (§3) the first demonstrably practical Byzantine consensus protocol supporting not only static consensus groups but also dynamic membership proportional to proof-of-work as in Bitcoin.
- We demonstrate experimentally (§4) that a strongly-consistent cryptocurrency can increase Bitcoin’s throughput by two orders of magnitude, with a trans-

action confirmation latency under one minute.

- We find through security analysis (§5) that ByzCoin can mitigate several known attacks on Bitcoin provided no attacker controls more than $\frac{1}{4}$ of hash power.

2 Background and Motivation

This section first outlines the three most relevant areas of prior work that ByzCoin builds on: cryptocurrencies such as Bitcoin and Bitcoin-NG, Byzantine fault tolerance (BFT) principles, and collective signing techniques.

2.1 Bitcoin and Variations

Bitcoin. At the core of Bitcoin [47] rests the so-called *blockchain*, a public, append-only database maintained by *miners* and serving as a global ledger of all transactions ever issued. Transactions are bundled into *blocks* and validated by a *proof-of-work*. A block is valid if its cryptographic hash has d leading zero bits, where the difficulty parameter d is adjusted periodically such that new blocks are mined about every ten minutes on average. Each block includes a Merkle tree [44] of new transactions to be committed, and a cryptographic hash chaining to the last valid block, thereby forming the blockchain. Upon successfully forming a new block with a valid proof-of-work, a miner broadcasts the new block to the rest of the miners, who (when behaving properly) accept the new block, if it extends a valid chain strictly longer than any they have already seen.

Bitcoin’s decentralized consensus and security derive from an assumption that a majority of the miners, measured in terms of *hash power* or ability to solve hash-based proof-of-work puzzles, follows these rules and always attempts to extend the longest existing chain. As soon as a quorum of miners with the majority of the network’s hash power approves a given block by mining on top of it, the block remains embedded in any future chain [29]. Bitcoin’s security is guaranteed by the fact that this majority will be extending the legitimate chain faster than any corrupt minority that might try to rewrite history or double-spend currency. However, Bitcoin’s consistency guarantee is only probabilistic, which leads to two fundamental problems.

First, multiple miners might find distinct blocks with the same parent before the network has reached consensus. Such a conflict is called a *fork*, an inconsistency that is temporarily allowed until one of the chains is extended yet again. Subsequently, all well-behaved miners on the shorter chain(s) switch to the new longest one. All transactions appearing only in the rejected block(s) are invalid and must be resubmitted for inclusion into the winning blockchain. This means that Bitcoin clients who want high certainty that a transaction is complete (e.g., that

they have irrevocably received a payment) must wait not only for the next block but for several blocks thereafter, thus increasing the time interval until a transaction can be considered complete. As a rule of thumb [47], a block is considered as permanently added to the blockchain after about 6 new blocks have been mined on top of it, for a confirmation latency of 60 minutes on average.

Second, the Bitcoin block size is currently limited to 1 MB. This limitation in turn results in an upper bound on the number of transactions per second (TPS) the Bitcoin network can handle, estimated to be an average of 7 TPS. For comparison, Paypal handles 500 TPS and VISA even 4000 TPS. An obvious solution to enlarge Bitcoin’s throughput is to increase the size of its blocks. Unfortunately, this solution also increases the probability of forks due to higher propagation delays and the risk of double-spending attacks [53, 30, 36]. Bitcoin’s liveness and security properties depend on forks being relatively rare. Otherwise, the miners would spend much of their effort trying to resolve multiple forks [31, 17], or in the extreme case, completely centralize Bitcoin [24].

Bitcoin-NG. Bitcoin-NG [24] makes the important observation that Bitcoin blocks serve two different purposes: (1) election of a leader who decides how to resolve potential inconsistencies, and (2) verification of transactions. Due to this observation, Bitcoin-NG proposes two different block types: *Keyblocks* are generated through mining with proof-of-work and are used to securely elect leaders, at a moderate frequency, such as every 10 minutes as in Bitcoin. *Microblocks* contain transactions, require no proof-of-work, and are generated and signed by the elected leader. This separation enables Bitcoin-NG to process many microblocks between the mining of two keyblocks, enabling transaction throughput to increase.

Bitcoin-NG, however, retains many drawbacks of Bitcoin’s consistency model. Temporary forks due to near-simultaneous keyblock mining, or deliberately introduced by selfish or malicious miners, can still throw the system into an inconsistent state for 10 minutes or more. Further, within any 10-minute window the current leader could still intentionally fork or rewrite history and invalidate transactions. If a client does not wait several tens of minutes (as in Bitcoin) for transaction confirmation, he is vulnerable to double-spend attacks by the current leader or by another miner who forks the blockchain. Although Bitcoin-NG includes disincentives for such behavior, these disincentives amount at most to the “mining value” of the keyblock (coinbase rewards and transaction fees): Thus, leaders are both able and have incentives to double-spend on higher-value transactions.

Consequently, although Bitcoin-NG permits higher transaction throughput, it does not solve Bitcoin’s con-

sistency weaknesses. Nevertheless, Bitcoin-NG’s decoupling of keyblocks from microblocks is an important idea that we build on in Section 3.6 to support high-throughput and low-latency transactions in ByzCoin.

2.2 Byzantine Fault Tolerance

The *Byzantine Generals’ Problem* [39, 49] refers to the situation where the malfunctioning of one or several components of a distributed system prevents the latter from reaching an agreement. Pease et al. [49] show that $3f + 1$ participants are necessary to be able to tolerate f faults and still reach consensus. The *Practical Byzantine Fault Tolerance (PBFT)* algorithm [14] was the first efficient solution to the Byzantine Generals’ Problem that works in weakly synchronous environments such as the Internet. PBFT offers both *safety* and *liveness* provided that the above bound applies, *i.e.*, that at most f faults among $3f + 1$ participants occur. PBFT triggered a surge of research on Byzantine replication algorithms with various optimizations and trade-offs [1, 16, 38, 32].

Every round of PBFT has three distinct phases. In the first, *pre-prepare* phase, the current primary node or *leader* announces the next record that the system should agree upon. On receiving this pre-prepare, every node validates the correctness of the proposal and multicasts a *prepare* message to the group. The nodes wait until they collect a quorum of $(2f + 1)$ prepare messages and publish this observation with a *commit* message. Finally, they wait for a quorum of $(2f + 1)$ commit messages to make sure that enough nodes have recorded the decision.

PBFT relies upon a correct leader to begin each round and proceeds if a two-thirds quorum exists; consequently, the leader is an attack target. For this reason PBFT has a view-change protocol that ensures liveness in the face of a faulty leader. All nodes monitor the leader’s actions and if they detect either malicious behavior or a lack of progress, initiate a view-change. Each node independently announces its desire to change leaders and stops validating the leader’s actions. If a quorum of $(2f + 1)$ nodes decides that the leader is faulty, then the next leader in a well-known schedule takes over.

PBFT has its limitations. First, it assumes a fixed, well-defined group of replicas, thus contradicting Bitcoin’s basic principle of being decentralized and open for anyone to participate. Second, each PBFT replica normally communicates directly with every other replica during each consensus round, resulting in $O(n^2)$ communication complexity: This is acceptable when n is typically 4 or not much more, but becomes impractical if n represents hundreds or thousands of Bitcoin nodes. Third, after submitting a transaction to a PBFT service, a client must communicate with a super-majority of the replicas in order to confirm the transaction has been com-

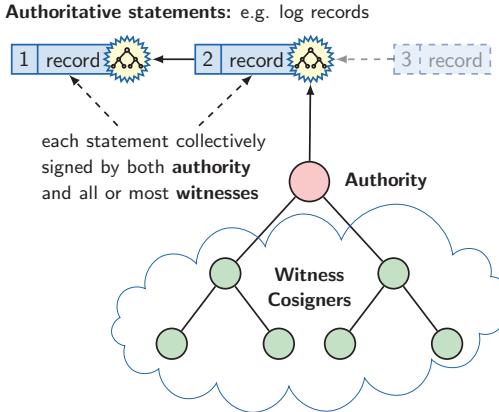


Figure 1: CoSi protocol architecture

mitted and to learn its outcome, making secure transaction verification unscalable.

2.3 Scalable Collective Signing

CoSi [54] is a protocol for scalable collective signing, which enables an authority or *leader* to request that statements be publicly validated and (*co-*)signed by a decentralized group of *witnesses*. Each protocol run yields a *collective signature* having size and verification cost comparable to an individual signature, but which compactly attests that both the leader and its (perhaps many) witnesses observed and agreed to sign the statement.

To achieve scalability, CoSi combines Schnorr multi-signatures [51] with communication trees that are long used in multicast protocols [13, 21, 55]. Initially, the protocol assumes that signature verifiers know the public keys of the leader and those of its witnesses, all of which combine to form a well-known aggregate public key. For each message to be collectively signed, the leader then initiates a CoSi four-phase protocol round that require two round-trips over the communication tree between the leader and its witnesses:

1. **Announcement:** The leader broadcasts an announcement of a new round down the communication tree. The announcement can optionally include the message M to be signed, otherwise M is sent in phase three.
2. **Commitment:** Each node picks a random secret and uses it to compute a Schnorr commitment. In a bottom-up process, each node obtains an aggregate Schnorr commitment from its immediate children, combines those with its own commitment, and passes a further-aggregated commitment up the tree.
3. **Challenge:** The leader computes a collective Schnorr challenge using a cryptographic hash function and broadcasts it down the communication tree, along with the message M to sign, if the latter has not already

been sent in phase one.

4. **Response:** Using the collective challenge, all nodes compute an aggregate response in a bottom-up fashion that mirrors the commitment phase.

The result of this four-phase protocol is the production of a standard Schnorr signature that requires about 64 bytes, using the Ed25519 elliptic curve [6], and that anyone can verify against the aggregate public key nearly as efficiently as the verification of an individual signature. Practical caveats apply if some witnesses are offline during the collective signing process: in this case the CoSi protocol can proceed, but the resulting signature grows to include metadata verifiably documenting which witnesses did and did not co-sign. We refer to the CoSi paper for details [54].

3 ByzCoin Design

This section presents ByzCoin with a step-by-step approach, starting from a simple “strawman” combination of PBFT and Bitcoin. From this strawman, we progressively address the challenges of determining consensus group membership, adapting Bitcoin incentives and mining rewards, making the PBFT protocol scale to large groups and handling block conflicts and selfish mining.

3.1 System Model

ByzCoin is designed for untrustworthy networks that can arbitrarily delay, drop, re-order or duplicate messages. To avoid the FLP impossibility [27], we assume the network has a weak synchrony property [14]. The Byz-Coin system is comprised of a set of N block miners that can generate key-pairs, but there is no trusted public-key infrastructure. Each node i has a limited amount of *hash power* that corresponds to the maximum number of block-header hashes the node can perform per second.

At any time t a subset of miners $M(t)$ is controlled by a malicious attacker and are considered faulty. Byzantine miners can behave arbitrarily, diverting from the protocol and colluding to attack the system. The remaining honest miners follow the prescribed protocol. We assume that the total hash power of all Byzantine nodes is less than $\frac{1}{4}$ of the system’s total hash power at any time, since proof-of-work-based cryptocurrencies become vulnerable to selfish mining attacks by stronger adversaries [25].

3.2 Strawman Design: PBFTCoin

For simplicity, we begin with PBFTCoin, an unrealistically simple protocol that naively combines PBFT with Bitcoin, then gradually refine it into ByzCoin.

For now, we simply assume that a group of $n = 3f + 1$ PBFT replicas, which we call *trustees*, has been fixed and

globally agreed upon upfront, and that at most f of these trustees are faulty. As in PBFT, at any given time, one of these trustees is the *leader*, who proposes transactions and drives the consensus process. These trustees collectively maintain a Bitcoin-like blockchain, collecting transactions from clients and appending them via new blocks, while guaranteeing that only one blockchain history ever exists and that it can never be rolled back or rewritten. Prior work has suggested essentially such a design [17, 19], though without addressing the scalability challenges it creates.

Under these simplifying assumptions, PBFTCoin guarantees safety and liveness, as at most f nodes are faulty and thus the usual BFT security bounds apply. However, the assumption of a fixed group of trustees is unrealistic for Bitcoin-like decentralized cryptocurrencies that permit open membership. Moreover, as PBFT trustees authenticate each other via non-transferable symmetric-key MACs, each trustee must communicate directly with most other trustees in every round, thus yielding $O(n^2)$ communication complexity.

Subsequent sections address these restrictions, transforming PBFTCoin into ByzCoin in four main steps:

1. We use Bitcoin’s proof-of-work mechanism to determine consensus groups dynamically while preserving Bitcoin’s defense against Sybil attacks.
2. We replace MAC-authenticated direct communication with digital signatures to make authentication transferable and thereby enabling sparser communication patterns that can reduce the normal case communication latency from $O(n^2)$ to $O(n)$.
3. We employ scalable collective signing to reduce per-round communication complexity further to $O(\log n)$ and reduce typical signature verification complexity from $O(n)$ to $O(1)$.
4. We decouple transaction verification from leader election to achieve a higher transaction throughput.

3.3 Opening the Consensus Group

Removing PBFTCoin’s assumption of a closed consensus group of trustees presents two conflicting challenges. On the one hand, conventional BFT schemes rely on a well-defined consensus group to guarantee safety and liveness. On the other hand, Sybil attacks [23] can trivially break any open-membership protocol involving security thresholds, such as PBFT’s assumption that at most f out of $3f + 1$ members are honest.

Bitcoin and many of its variations employ a mechanism already suited to this problem: proof-of-work mining. Only miners who have dedicated resources are allowed to become a member of the consensus group. In refining PBFTCoin, we adapt Bitcoin’s proof-of-work mining into a *proof-of-membership* mechanism. This

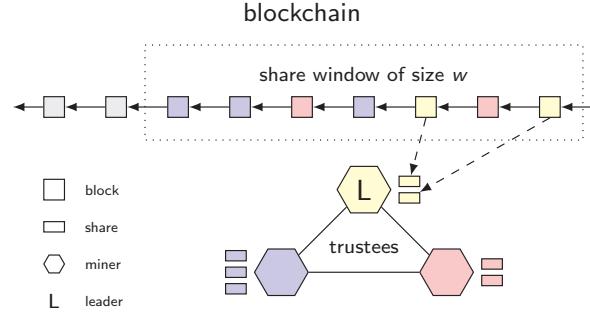


Figure 2: Valid shares for mined blocks in the blockchain are credited to miners

mechanism maintains the “balance of power” within the BFT consensus group over a given fixed-size sliding *share window*. Each time a miner finds a new block, it receives a *consensus group share*, which proves the miner’s membership in the group of trustees and moves the share window one step forward. Old shares beyond the current window expire and become useless for purposes of consensus group membership. Miners holding no more valid shares in the current window lose their membership in the consensus group, hence they are no longer allowed to participate in the decision-making.

At a given moment in time, each miner wields “voting power” of a number of shares equal to the number of blocks the miner has successfully mined within the current window. Assuming collective hash power is relatively stable, this implies that within a window, each active miner wields a number of shares statistically proportionate to the amount of hash power that the miner has contributed during this time period.

The size w of the share window is defined by the average block-mining rate over a given time frame and influences certain properties such as the resilience of the protocol to faults. For example, if we assume an average block-mining rate of 10 minutes and set the duration of the time frame to one day (or one week), then $w = 144$ ($w = 1008$). This mechanism limits the membership of miners to recently active ones, which prevents the system from becoming unavailable due to too many trustees becoming inactive over time, or from miners aggregating many shares over an extended period and threatening the balance in the consensus group. The relationship between blocks, miners and shares is illustrated in Fig. 2.

Mining Rewards and Transaction Fees. As we can no longer assume voluntary participation as in PBFTCoin’s closed group of trustees, we need an incentive for nodes to obtain shares in the consensus group and to remain active. For this purpose, we adopt Bitcoin’s ex-

isting incentives of mining rewards and transaction fees. But instead of these rewards all going to the miner of the most recent block we split a new block’s rewards and fees across all members of the current consensus group, in proportion to the number of shares each miner holds. As a consequence, the more hash power a miner has devoted within the current window, hence the more shares the miner holds, the more revenue the miner receives during payouts in the current window. This division of rewards also creates incentives for consensus group members to remain live and participate, because they receive their share of the rewards for new blocks only if they continually participate, in particular contributing to the prepare and commit phases of each BFT consensus round.

3.4 Replacing MACs by Digital Signatures

In our next refinement step towards ByzCoin, we tackle the scalability challenge resulting from PBFT’s typical communication complexity of $O(n^2)$, where n is the group size. PBFT’s choice of MAC-authenticated all-to-all communication was motivated by the desire to avoid public-key operations on the critical transaction path. However, the cost for public-key operations has decreased due to well-optimized asymmetric cryptosystems [6], making those costs less of an issue.

By adopting digital signatures for authentication, we are able to use sparser and more scalable communication topologies, thus enabling the current leader to collect and distribute third-party verifiable evidence that certain steps in PBFT have succeeded. This removes the necessity for all trustees to communicate directly with each other. With this measure we can either enable the leader to collect and distribute the digital signatures, or let nodes communicate in a chain [32], reducing the normal-case number of messages from $O(n^2)$ to $O(n)$.

3.5 Scalable Collective Signing

Even with signatures providing transferable authentication, the need for the leader to collect and distribute – and for all nodes to verify – many individual signatures per round can still present a scalability bottleneck. Distributing and verifying tens or even a hundred individual signatures per round might be practical. If we want consensus groups with a thousand or more nodes, however (e.g., representing all blocks successfully mined in the past week), it is costly for the leader to distribute 1000 digital signatures and wait for everyone to verify them. To tackle this challenge, we build on the CoSi protocol [54] for collective signing. CoSi does not directly implement consensus or BFT, but it offers a primitive that the leader in a BFT protocol can use to collect and aggregate prepare and commit messages during PBFT rounds.

We implement a single ByzCoin round by using two sequential CoSi rounds initiated by the current leader (*i.e.*, the owner of the current view). The leader’s announcement of the first CoSi round (phase 1 in Section 2.3) implements the *pre-prepare* phase in the standard PBFT protocol (Section 2.2). The collective signature resulting from this first CoSi round implements the PBFT protocol’s *prepare* phase, in which the leader obtains attestations from a two-thirds super-majority quorum of consensus group members that the leader’s proposal is safe and consistent with all previously-committed history.

As in PBFT, this prepare phase ensures that a proposal *can be* committed consistently, but by itself it is insufficient to ensure that the proposal *will be* committed. The leader and/or some number of other members could fail before a super-majority of nodes learn about the successful prepare phase. The ByzCoin leader therefore initiates a second CoSi round to implement the PBFT protocol’s *commit* phase, in which the leader obtains attestations from a two-thirds super-majority that all the signing members witnessed the successful result of the prepare phase and made a positive commitment to remember the decision. This collective signature, resulting from this second CoSi round, effectively attests that a two-thirds super-majority of members not only considers the leader’s proposal “safe” but promises to remember it, indicating that the leader’s proposal is fully committed.

In cryptocurrency terms, the collective signature resulting from the prepare phase provides a proof-of-acceptance of a proposed block of transactions. This block is not yet committed, however, since a Byzantine leader that does not publish the accepted block could double-spend by proposing a conflicting block in the next round. In the second CoSi commit round, the leader announces the proof-of-acceptance to all members, who then validate it and collectively sign the block’s hash to produce a collective commit signature on the block. This way a Byzantine leader cannot rewrite history or double-spend, because by counting arguments at least one honest node would have to sign the commit phase of both histories, which an honest node by definition would not do.

The use of CoSi does not affect the fundamental principles or semantics of PBFT but improves its scalability and efficiency in two main ways. First, during the commit round where each consensus group member must verify that a super-majority of members have signed the prior prepare phase, each participant generally needs to receive only an $O(1)$ -size rather than $O(n)$ -size message, and to expend only $O(1)$ rather than $O(n)$ computation effort by verifying a single collective signature instead of n individual ones. This benefit directly increases the scalability and reduces the bandwidth and computation costs of consensus rounds themselves.

A second benefit is that after the final CoSi commit round has completed, the final resulting collective commitment signature serves as a typically $O(1)$ -size proof, which anyone can verify in $O(1)$ computation time that a given block – hence any transaction within that block – has been irreversibly committed. This secondary scalability-benefit might in practice be more important than the first, because it enables “light clients” who neither mine blocks nor store the entire blockchain history to verify quickly and efficiently that a transaction has committed, without requiring active communication with or having to trust any particular full node.

3.6 Decoupling Transaction Verification from Leader Election

Although ByzCoin so far provides a scalable guarantee of strong consistency, thus ensuring that clients need to wait only for the next block rather than the next several blocks to verify that a transaction has committed, the time they still have to wait *between* blocks can, nevertheless, be significant: *e.g.*, up to 10 minutes using Bitcoin’s difficulty tuning scheme. Whereas ByzCoin’s strong consistency might in principle make it “safe” from a consistency perspective to increase block mining rate, doing so could still exacerbate liveness and other performance issues, as in Bitcoin [47]. To enable lower client-perceived transaction latency, therefore, we build on the idea of Bitcoin-NG [24] to decouple the functions of transaction verification from block mining for leader election and consensus group membership.

As in Bitcoin-NG, we use two different kinds of blocks. The first, *microblocks* or transaction blocks, represent transactions to be stored and committed. The current leader creates a new microblock every few seconds, depending on the size of the block, and uses the CoSi-based PBFT protocol above to commit and collectively sign it. The other type of block, *keyblocks*, are mined via proof-of-work as in Bitcoin and serve to elect leaders and create shares in ByzCoin’s group membership protocol as discussed earlier in Section 3.3. As in Bitcoin-NG, this decoupling allows the current leader to propose and commit many microblocks that contain many smaller batches of transactions, within one ≈ 10 -minute keyblock mining period. Unlike Bitcoin-NG, in which a malicious leader could rewrite history or double-spend within this period until the next keyblock, ByzCoin ensures that each microblock is irreversibly committed regardless of the current leader’s behavior.

In Bitcoin-NG one blockchain includes both types of blocks, which introduces a race condition for miners. As microblocks are created, the miners have to change the header of their keyblocks to mine on top of the latest microblock. In ByzCoin, in contrast, the blockchain

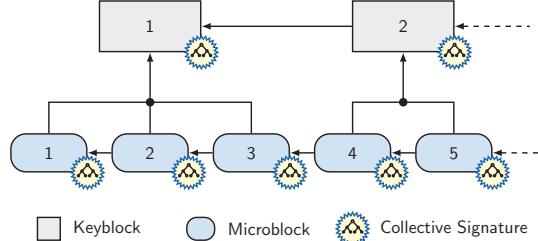


Figure 3: ByzCoin blockchain: Two parallel chains store information about the leaders (keyblocks) and the transactions (microblocks)

becomes two separate parallel blockchains, as shown in Fig. 3. The main blockchain is the keyblock chain, consisting of all mined blocks. The microblock chain is a secondary blockchain that depends on the primary to identify the era in which every microblock belongs to, *i.e.*, which miners are authoritative to sign it and who is the leader of the era.

Microblocks. A microblock is a simple block that the current consensus group produces every few seconds to represent newly-committed transactions. Each microblock includes a set of transactions and a collective signature. Each microblock also includes hashes referring to the previous microblock and keyblock: the former to ensure total ordering, and the latter indicating which consensus group window and leader created the microblock’s signature. The microblock’s hash is collectively signed by the corresponding consensus group.

Keyblocks. Each keyblock contains a proof-of-work, which is used to determine consensus group membership via the sliding-window mechanism discussed earlier, and to pay signers their rewards. Each newly-mined keyblock defines a new consensus group, and hence a new set of public keys with which the next era’s microblocks will be collectively signed. Since each successive consensus group differs from the last in at most one member, PBFT ensures the microblock chain’s consistency and continuity across this group membership change provided at most f out of $3f + 2$ members are faulty.

Bitcoin-NG relies on incentives to discourage the next leader from accidentally or maliciously “forgetting” a prior leader’s microblocks. In contrast, the honest supermajority in a ByzCoin consensus group will refuse to allow a malicious or amnesiac leader to extend any but the most recently-committed microblock, regardless of which (current or previous) consensus group committed it. Thus, although competing keyblock conflicts may still appear, these “forks” cannot yield an inconsistent microblock chain. Instead, a keyblock conflict can at

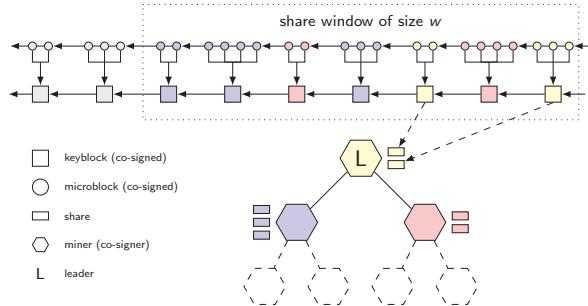


Figure 4: Overview of the full ByzCoin design

worst temporarily interrupt the PBFT protocol’s liveness, until it is resolved as mentioned in Section 3.6.1.

Decoupling transaction verification from leader election and consensus group evolution in this way brings the overall ByzCoin architecture to completion, as illustrated in Fig. 4. Subsequent sections discuss further implications and challenges this architecture presents.

3.6.1 Keyblock Conflicts and Selfish Mining

PBFT’s strong consistency by definition does not permit inconsistencies such as forks in the microblock chain. The way the miners collectively decide how to resolve keyblock conflicts, however, can still allow selfish mining [25] to occur as in Bitcoin. Worse, if the miners decide randomly to follow one of the two blocks, then keyblock forks might frequently lead to PBFT liveness interruptions as discussed above, by splitting miners “too evenly” between competing keyblocks. Another approach to deciding between competing keyblocks is to impose a deterministic priority function on their hash values, such as “smallest hash wins.” Unfortunately, this practice can encourage selfish mining.

One way to break a tie without helping selfish miners, is to increase the entropy of the output of the deterministic prioritization function. We implement this idea using the following algorithm. When a miner detects a keyblock fork, it places all competing blocks’ header hashes into a sorted array, from low to high hash values. The array itself is then hashed, and the final bit(s) of this hash determine which keyblock wins the fork.

This solution, shown in Fig. 5, also uses the idea of a deterministic function applied to the blocks, thus requiring no voting. Its advantage is that the input of the hash function is partially unknown before the fork occurs, thus the entropy of the output is high and difficult for an attacker to be able to optimize. Given that the search space for a possible conflict is as big as the search space for a new block, trying to decide if a block has better than 50% probability of winning the fork is as hard as finding a new block.

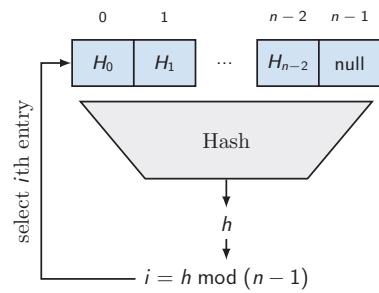


Figure 5: Deterministic fork resolution in ByzCoin

3.6.2 Leader Election and PBFT View Changes

Decoupling transaction verification from the block-mining process comes at a cost. So far we have assumed, as in PBFT, that the leader remains fixed unless he fails. If we keep this assumption, then this leader gains the power of deciding which transactions are verified, hence we forfeit the fairness-enforcing benefit of Bitcoin’s leader election. To resolve this issue, every time a keyblock is signed, ByzCoin forces a mandatory PBFT view-change to the keyblock’s miner. This way the power of verifying transactions in blocks is assigned to the rightful miner, who has an era of microblock creation from the moment his keyblock is signed until the next keyblock is signed.

When a keyblock conflict occurs, more than one such “mandatory” view-change occurs, with the successful miners trying to convince other participants to adopt their keyblock and its associated consensus group. For example, in a keyblock fork, one of the two competing keyblocks wins the resolution algorithm described above. However, if the miner of the “losing” block races to broadcast its keyblock and more than 33% honest miners have already committed to it before learning about the competing keyblock, then the “winning” miner is too late and the system either commits to the first block or (in the worst case) loses liveness temporarily as discussed above. This occurs because already-committed miners will not accept a mandatory view-change except to a keyblock that represents their committed state and whose microblock chain extends all previously-committed microblocks. Further analysis of how linearizability is preserved across view-changes may be found in the original PBFT paper [14].

3.6.3 Tree Creation in ByzCoin

Once a miner successfully creates a new keyblock, he needs to form a CoSi communication tree for collective signing, with himself as the leader. If all miners individually acknowledge this keyblock to transition to

the next view, this coordination normally requires $O(N)$ messages. To avoid this overhead at the beginning of each keyblock round, the miners autonomously create the next round’s tree bottom-up during the previous keyblock round. This can be done in $O(1)$ by using the blockchain as an array that represents a full tree.

This tree-building process has three useful side-effects. First, the previous leader is the first to get the new block, hence he stops creating microblocks and wasting resources by trying to sign them. Second, in the case of a keyblock conflict, potential leaders use the same tree, and the propagation pattern is the same; this means that all nodes will learn and decide on the conflict quickly. Finally, in the case of a view change, the new view will be the last view that worked correctly. So if the leader of the keyblock i fails, the next leader will again be the miner of keyblock $i - 1$.

3.7 Tolerating Churn and Byzantine Faults

In this section we discuss the challenges of fault tolerance in ByzCoin, particularly tree failures and maximum tolerance for Byzantine faults.

3.7.1 Tree Fault Tolerance

In CoSi, there are multiple different mechanisms that allow substantial fault-tolerance. Furthermore the strict membership requirements and the frequent tree changes of ByzCoin increase the difficulty for a malicious attacker with less than around 25% of the total hash power to compromise the system. A security analysis, however, must assume that a Byzantine adversary is able to get the correct nodes of the ByzCoin signing tree so that it can compromise the liveness of the system by a simple DoS.

To mitigate this risk, we focus on recent Byzantine fault tolerance results [32], modifying ByzCoin so that the tree communication pattern is a normal-case performance optimization that can withstand most malicious attacks. But when the liveness of the tree-based ByzCoin is compromised, the leader can return to non-tree-based communication until the end of his era.

The leader detects that the tree has failed with the following mechanism: After sending the block to his children, the leader starts a timer that expires before the view-change timer. Then he broadcasts the hash of the block he proposed and waits. When the nodes receive this message they check if they have seen the block and either send an ACK or wait until they see the block and then send the ACK. The leader collects and counts the ACKs, to detect if his block is rejected simply because it never reaches the witnesses. If the timer expires or a block rejection arrives before he receives two-thirds of the ACKs, the leader knows that the tree has failed and

reverts to a flat ByzCoin structure before the witnesses assume that he is faulty.

As we show in Section 4, the flat ByzCoin structure can still quickly sign keyblocks for the day-long window (144 witnesses) while maintaining a throughput higher than Bitcoin currently supports. Flat ByzCoin is more robust to faults, but increases the communication latency back to $O(n)$. Furthermore, if all faults ($\lfloor \frac{N}{3} \rfloor$) are consecutive leaders, this can lead back to a worst case $O(n^2)$ communication latency.

3.7.2 Membership Churn and BFT

After a new leader is elected, the system needs to ensure that the first microblock of the new leader points to the last microblock of the previous leader. Having $2f + 1$ supporting votes is not enough. This occurs because there is the possibility than an honest node lost its membership when the new era started. Now in the worst case, the system has f Byzantine nodes, f honest nodes that are up to date, f slow nodes that have a stale view of the blockchain, and the new leader that might also have a stale view. This can lead to the leader proposing a new microblock, ignoring some signed microblocks and getting $2f + 1$ support (stale+Byzantine+his own). For this reason, the first microblock of an era needs $2f + 2$ supporting signatures. If the leader is unable to obtain them, this means that he needs to synchronize with the system, *i.e.*, he needs to find the latest signed microblock from the previous roster. He asks all the nodes in his roster, plus the node that lost its membership, to sign a latest-checkpoint message containing the hash of the last microblock. At this point in time, the system has $3f + 2$ ($3f + 1$ of the previous roster plus the leader) members and needs $2f + 1$ honest nodes to verify the checkpoint, plus an honest leader to accept it (a Byzantine leader will be the $f + 1$ fault and compromise liveness). Thus, ByzCoin can tolerate f fails in a total of $3f + 2$ nodes.

4 Performance Evaluation

In this section we discuss the evaluation of the ByzCoin prototype and our experimental setup. The main question we want to evaluate is whether ByzCoin is usable in practice without incurring large overheads. In particular we focus on consensus latency and transaction throughput for different parameter combinations.

4.1 Prototype Implementation

We implemented ByzCoin in Go¹ and made it publicly available on GitHub.² ByzCoin’s consensus mecha-

¹<https://golang.org>

²<https://github.com/DeDiS/Cothority>

nism is based on the CoSi protocol with Ed25519 signatures [6] and implements both flat- and tree-based collective signing layouts as described in Section 3. For comparison, we also implemented a conventional PBFT consensus algorithm with the same communication patterns as above and a consensus algorithm that uses individual signatures and tree-based communication.

To simulate consensus groups of up to 1008 nodes, we oversubscribed the available 36 physical machines (see below) and ran up to 28 separate ByzCoin processes on each server. Realistic wide-area network conditions are mimicked by imposing a round-trip latency of 200 ms between any two machines and a link bandwidth of 35 Mbps per simulated host. Note that this simulates only the connections between miners of the consensus group and not the full Bitcoin network. Full nodes and clients are not part of the consensus process and can retrieve signed blocks only after consensus is reached. Since Bitcoin currently is rather centralized and has only a few dozen mining pools [3], we assume that if/when decentralization happens, all miners will be able to support these rather constrained network requirements.

The experimental data to form microblocks was taken by ByzCoin clients from the actual Bitcoin blockchain. Both micro- and keyblocks are fully transmitted and collectively signed through the tree and are returned to the clients upon request together with the proof. Verification of block headers is implemented but transaction verification is only emulated to avoid further measurement distortion through oversubscribed servers. A similar practice is used in Shadow Bitcoin [45]. We base our emulation both on measurements [31] of the average block-verification delays (around 200 ms for 500 MB blocks) and on the claims of Bitcoin developers [8] that as far as hardware is concerned Bitcoin can easily verify 4000 TPS. We simulate a linear increase of this delay proportional to the number of transactions included in the block. Because of the communication pattern of ByzCoin, the transaction-verification cost delays only the leaf nodes. By the time the leaf nodes finish the block verification and send their vote back to their parents, all other tree nodes should have already finished the verification and can immediately proceed. For this reason the primary delay factor is the transmission of the blocks that needs to be done $\log N$ sequential times.

We ran all our experiments on DeterLab [22] using 36 physical machines, each having four Intel E5-2420 v2 CPUs and 24 GB RAM and being arranged in a star-shaped virtual topology.

4.2 Consensus Latency

The first two experiments focus on how microblock commitment latency scales with consensus group size and

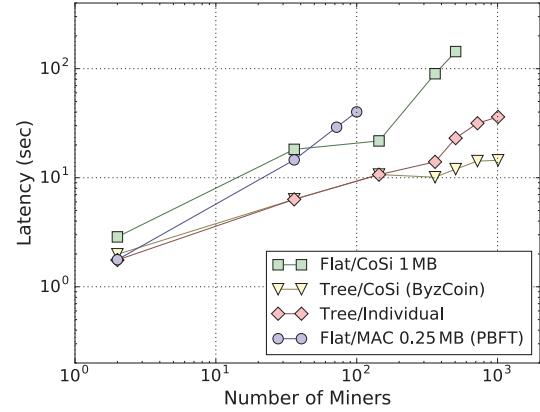


Figure 6: Influence of the consensus group size on the consensus latency

with number of transactions per block.

4.2.1 Consensus Group Size Comparison

This experiment focuses on the scalability of ByzCoin’s BFT protocol in terms of the consensus group size. The number of unique miners participating in a consensus group is limited by the number of membership shares in the window (Section 3.3), but can be smaller if some miners hold multiple shares (*i.e.*, successfully mined several blocks) within the same window.

We ran experiments for Bitcoin’s maximum block size (1 MB) with a variable number of participating hosts. Every time we increased the number of hosts, we also increased the servers’ bandwidth so that the available bandwidth per simulated host remained constant (35 Mbps). For the PBFT simulation, the 1 MB block was too big to handle, thus the PBFT line corresponds to a 250 KB block size.

As Fig. 6 shows, the simple version of ByzCoin achieves acceptable latency, as long as the consensus group size is less than 200. After this point the cost for the leader to broadcast the block to everyone incurs large overheads. On the contrary, the tree-based ByzCoin scales well, since the same 1 MB block for 1008 nodes suffers signing latency less than the flat approach for 36 nodes. Adding 28 times more nodes (from 36 to 1008) causes a latency increase close to a factor 2 (from 6.5 to 14 seconds). The basic PBFT implementation is quite fast for 2 nodes but scales poorly (40 seconds for 100 nodes), whereas the tree-based implementation with individual signatures performs the same as ByzCoin for up to 200 hosts. If we aim for the higher security level of 1008 nodes, however, then ByzCoin is 3 times faster.

Fig. 7 shows the performance cost of keyblock sign-

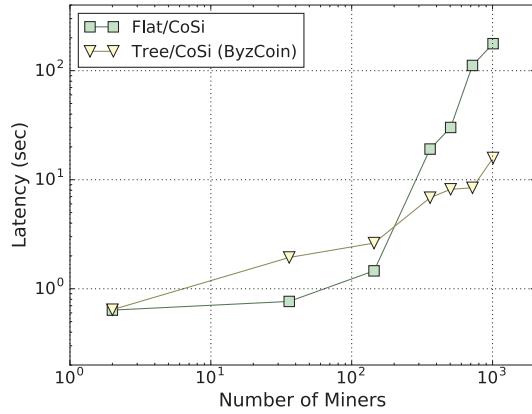


Figure 7: Keyblock signing latency

ing. The flat variant outperforms the tree-based version when the number of hosts is small since the blocks have as many transactions as there are hosts and thus are small themselves. This leads to a fast transmission even in the flat case and the main overhead comes from the block propagation latency, which scales with $O(\log N)$ in the tree-based ByzCoin variant.

4.2.2 Block Size Comparison

The next experiment analyzes how different block sizes affect the scalability of ByzCoin. We used a constant number of 144 hosts for all implementations. Once again, PBFT was unable to achieve acceptable latency with 144 nodes, thus we ran it with 100 nodes only.

Fig. 8 shows the average latency of the consensus mechanism, determined over 10 blocks when their respective sizes increase. As in the previous section we see that the flat implementation is acceptable for a 1 MB block, but when the block increases to 2 MB the latency quadruples. This outcome is expected as the leader's link saturates when he tries to send 2 MB messages to every participating node. In contrast ByzCoin scales well because the leader outsources the transmission of the blocks to other nodes and contacts only his children. The same behavior is observed for the algorithm that uses individual signatures and tree-based communication, which shows that the block size has no negative effect on scalability when a tree is used. Finally, we find that PBFT is fast for small blocks, but the latency rapidly increases to 40 seconds for 250 KB blocks.

ByzCoin's signing latency for a 1 MB block is close to 10 seconds, which should be small enough to make the need for 0-confirmation transactions almost disappear. Even for a 32 MB block (≈ 66000 transactions) the delay is much lower (around 90 seconds) than the ≈ 10 minutes required by Bitcoin.

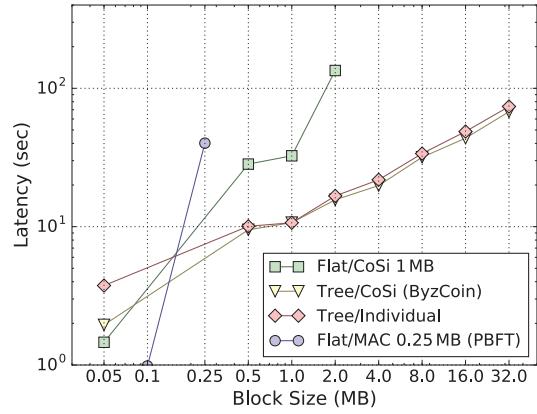


Figure 8: Influence of the block size on the consensus latency

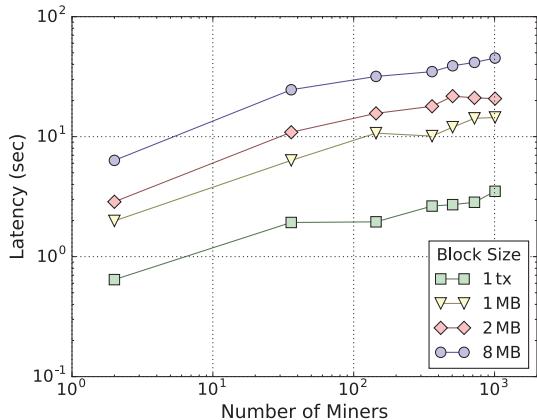


Figure 9: Influence of the consensus group size on the block signing latency

minutes required by Bitcoin.

Fig. 9 demonstrates the signing latency of various blocks sizes on tree-based ByzCoin. Signing one-transaction blocks takes only 3 seconds even when 1008 miners co-sign it. For bigger blocks, we have included Bitcoin's current maximum block size of 1 MB along with the proposed limits of 2 MB in Bitcoin Classic and 8 MB in Bitcoin Unlimited [2]. As the graph shows, 1 MB and 2 MB blocks scale linearly in number of nodes at first but after 200 nodes, the propagation latency is higher than the transmission of the block, hence the latency is close to constant. For 8 MB blocks, even with 1008 the signing latency increases only linearly.

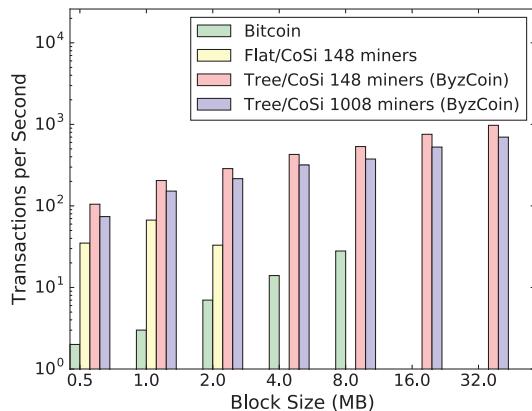


Figure 10: Throughput of ByzCoin

4.3 Transaction Throughput

In this experiment, we investigate the maximum throughput in terms of transactions per second (TPS) that ByzCoin can achieve, and show how Bitcoin could improve its throughput by adopting a ByzCoin-like deployment model. We tested ByzCoin versions with consensus group sizes of 144 and 1008 nodes, respectively. Note that performance-wise this resembles the worst case scenario since the miner-share ratio is usually not 1:1 as miners in the consensus group are allowed to hold multiple shares, as described in Section 3.3.

Analyzing Fig. 10 shows that Bitcoin can increase its overall throughput by more than one order of magnitude through adoption of a flat ByzCoin-like model, which separates transaction verification and block mining and deals with forks via strong consistency. Furthermore, the 144 node configuration can achieve close to 1000 TPS, which is double the throughput of Paypal, and even the 1008-node roster achieves close to 700 TPS. Even when the tree fails, the system can revert back to 1 MB microblocks on the flat and more robust variant of ByzCoin and still have a throughput ten times higher than the current maximum throughput of Bitcoin.

In both Figs. 8 and 10, the usual trade-off between throughput and latency appears. The system can work with 1–2 MB microblocks when the load is normal and then has a latency of 10–20 seconds. If an overload occurs, the system adaptively changes the block size to enable higher throughput. We note that this is not the case in the simple ByzCoin where 1 MB microblocks have optimal throughput and acceptable latency.

5 Security Analysis

In this section, we conduct a preliminary, informal security analysis of ByzCoin, and discuss how its consensus mechanism can mitigate or eliminate some known attacks against Bitcoin.

5.1 Transaction Safety

In the original Bitcoin paper [47], Nakamoto models Bitcoin’s security against transaction double spending attacks as in a Gambler’s Ruin Problem. Furthermore, he models the progress an attacker can make as a Poisson distribution and combines these two models to reach Eq. (1). This equation calculates the probability of a successful double spend after z blocks when the adversary controls q computing power.

$$P = 1 - \sum_{k=0}^z \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - \left(\frac{q}{p}\right)^{(z-k)}\right) \quad (1)$$

In Figs. 11 and 12 we compare the relative safety of a transaction over time in Bitcoin³ versus ByzCoin. Fig. 11 shows that ByzCoin can secure a transaction in less than a minute, because the collective signature guarantees forward security. On the contrary, Bitcoin’s transactions need hours to be considered fully secured from a double-spending attempt. Fig. 12 illustrates the required time from transaction creation to the point where a double spending attack has less than 0.1% chance of success. ByzCoin incurs a latency of below one minute to achieve the above security, which boils down to the time the systems needs to produce a collectively signed microblock. Bitcoin on the other hand needs several hours to reach the same guarantees. Note that this graph does not consider other advanced attacks, such as eclipse attacks [34], where Bitcoin offers no security for the victim’s transactions.

5.2 Proof-of-Membership Security

The security of ByzCoin’s proof-of-membership mechanism can be modeled as a random sampling problem with two possible independent outcomes (honest, Byzantine). The probability of picking a Byzantine node (in the worst case) is $p = 0.25$ and the number of tries corresponds to the share window size w . In this setting, we are interested in the probability that the system picks less than $c = \lfloor \frac{w}{3} \rfloor$ Byzantine nodes as consensus group members and hence guarantees safety. To calculate this probability, we use the cumulative binomial distribution where X is the random variable that represents the number of times we pick a Byzantine node:

³Based on data from <https://blockchain.info>.

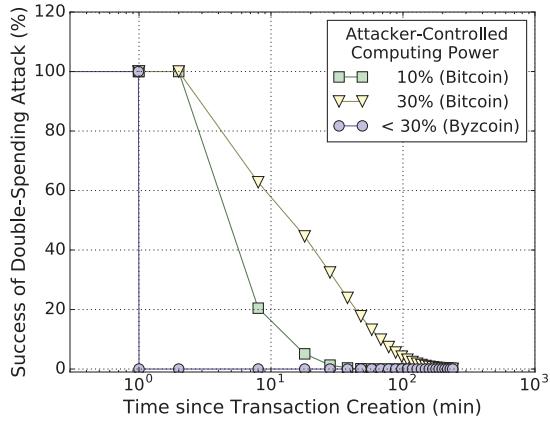


Figure 11: Successful double-spending attack probability

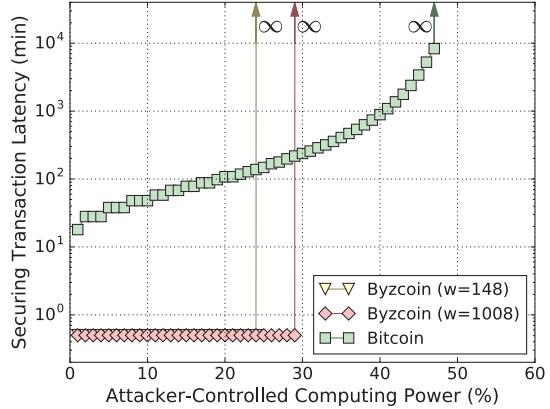


Figure 12: Client-perceived secure transaction latency

$$P[X \leq c] = \sum_{k=0}^c \binom{w}{k} p^k (1-p)^{w-k} \quad (2)$$

Table 1 displays the results for the evaluation of Eq. (2) for various window sizes w both in the common threat model where an adversary controls up to 25% hash power and in the situation where the system faces a stronger adversary with up to 30% computing power. The latter might temporarily occur due to hash power variations and resource churn.

Table 1: Expected proof-of-membership security levels

$p \mid w$	12	100	144	288	1008	2016
0.25	0.842	0.972	0.990	0.999	0.999	1.000
0.30	0.723	0.779	0.832	0.902	0.989	0.999

At this point, recall that w specifies the number of

available shares and not necessarily the number of actual miners as each member of the consensus group is allowed to hold multiple shares. This means that the number of available shares gives an upper bound on the latency of the consensus mechanism with the worst case being that each member holds exactly one share.

In order to choose a value for w appropriately one must take into account not only consensus latency and the desired security level (ideally $\geq 99\%$) but also the increased chance for resource churn when values of w become large. From a security perspective the results of Table 1 suggest that the share window size should not be set to values lower than $w = 144$. Ideally, values of $w = 288$ and above should be chosen to obtain a reasonable security margin and, as demonstrated in Section 4, values up to $w = 1008$ provide excellent performance numbers.

Finally, care should be taken when bootstrapping the protocol, as for small values of w there is a high probability that a malicious adversary is able to take over control. For this reason we suggest that ByzCoin starts with vanilla Nakamoto consensus and only after w keyblocks are mined the ByzCoin consensus is activated.

5.3 Defense Against Bitcoin Attacks

0-confirmation Double-Spend Attacks. Race [35] and Finney [26] attacks belong to the family of 0-confirmation double-spend attacks which might affect traders that provide real-time services to their clients. In such scenarios the time between exchange of currency and goods is usually short because traders often cannot afford to wait an extended period of time (10 or more minutes) until a transaction they received can be considered indeed confirmed.

ByzCoin can mitigate both attacks by putting the merchant’s transaction in a collectively signed microblock whose verification latency is in the order of a few seconds up to a minute. If this latency is also unacceptable, then he can send a single transaction for signing, which will cost more, but is secured in less than 4 seconds.

N-confirmation Double-Spend Attacks. The assumption underlying this family of attacks [7] is that, after receiving a transaction for a trade, a merchant waits $N - 1$ additional blocks until he concludes the interaction with his client. At this point, a malicious client creates a new double-spending transaction and tries to fork the blockchain, which has a non-negligible success-probability if the adversary has enough hash power. For example, if $N = 3$ then an adversary holding 10% of the network’s hash power has a 5% success-chance to mount the above attack [47].

In ByzCoin the merchant would simply check the collective signature of the microblock that includes the

transaction, which allows him to verify that it was accepted by a super-majority of the network. Afterwards the attacker cannot succeed in forking the blockchain as the rest of the signers will not accept his new block. Even if the attacker is the leader, the proposed microblock will be rejected, and a view change will occur.

Eclipse and Delivery-Tampering Attacks. In an eclipse attack [34] it is assumed that an adversary controls a sufficiently large number of connections between the victim and the Bitcoin network. This enables the attacker to mount attacks such as 0- and N-confirmation double-spends with an ever increasing chance of success the longer the adversary manages to keep his control over the network. Delivery-tampering attacks [31] exploit Bitcoin’s scalability measures to delay propagation of blocks without causing a network partition. This allows an adversary to control information that the victim receives and simplifies to mount 0- and 1-confirmation double-spend attacks as well as selfish-mining.

While ByzCoin does not prevent an attacker from eclipsing a victim or delaying messages in the peer-to-peer network, its use of collective signatures in transaction commitment ensure that a victim cannot be tricked into accepting an alternate attacker-controlled transaction history produced in a partitioned network fragment.

Selfish and Stubborn Mining Attacks. Selfish mining [25] allows a miner to increase his profit by adding newly mined blocks to a hidden blockchain instead of instantly broadcasting them. This effect can be further amplified if the malicious miner has good connectivity to the Bitcoin network. The authors of selfish mining propose a countermeasure that thwarts the attack if a miner has less than 25% hash power under normal circumstances or less than 33% in case of an optimal network. Stubborn mining [48] further generalizes the ideas behind selfish mining and combines it with eclipse attacks in order to increase the adversary’s revenue.

In ByzCoin, these strategies are ineffective as forks are instantly resolved in a deterministic manner, hence building a hidden blockchain only wastes resources and minimizes revenue. Another approach to prevent the above attacks would be to include bias-resistant public randomness [40] in every keyblock. This way even if an attacker gains control over the consensus mechanism (*e.g.*, by having > 33% hash power) he would still be unable to mine hidden blocks. We leave exploring this approach for future research.

Transaction Censorship. In Bitcoin-NG, a malicious leader can censor transactions for the duration of his epoch(s). The same applies for ByzCoin. However, as

not every leader is malicious, the censored transactions are only delayed and will be processed eventually by the next honest leader. ByzCoin can improve on this, as the leader’s actions are double-checked by all the other miners in the consensus group. A client can announce his censored transaction just like in classic PBFT; this will indicate a potential leader fault and will either stop censorship efforts or lead to a view-change to remove the malicious leader. Finally, in Bitcoin(-NG) a miner can announce his intention to fork over a block that includes a transaction, giving an incentive to other miners to exclude this transaction. In ByzCoin using fork-based attacks to censor transactions is no longer possible due to ByzCoin’s deterministic fork resolution mechanism. An attacker can therefore only vote down a leader’s proposals by refusing to co-sign. This is also a limitation, however, as an adversary who controls more than 33% of the shares (Section 7) deny service and can censor transactions for as long as he wants.

6 Related Work

ByzCoin and Bitcoin [47] share the same primary objective: implement a state machine replication (SMR) system with open membership [9, 29]. Both therefore differ from more classic approaches to Byzantine fault-tolerant SMRs with static or slowly changing consensus groups such as PBFT [14], Tendermint [10], or Hyperledger [42].

Bitcoin has well-known performance shortcomings; there are several proposals [41, 57] on how to address these. The GHOST protocol [53] changes the chain selection rule when a fork occurs. While Bitcoin declares the fork with the most proof-of-work as the new valid chain, GHOST instead chooses the entire subtree that received the most computational effort. Put differently, the subtree that was considered correct for the longest time will have a high possibility of being selected, making an intentional fork much harder. One limitation of GHOST is that no node will know the full tree, as invalid blocks are not propagated. While all blocks could be propagated, this makes the system vulnerable to DoS attacks since an adversary can simply flood the network with low-difficulty blocks.

Off-chain transactions, an idea that originated from the two-point channel protocol [33], are another alternative to improve latency and throughput of the Bitcoin network. Other similar proposals include the Bitcoin Lightning Network [50] and micro-payment channels [20], which allow transactions without a trusted middleman. They use contracts so that any party can generate proof-of-fraud on the main blockchain and thereby deny revenue to an attacker. Although these systems enable faster cryptocurrencies, they do not address the core problem

of scaling SMR systems, thus sacrificing the open and distributed nature of Bitcoin. Finally, the idea behind sidechains [5] is to connect multiple chains with each other and enable the transfer of Bitcoins from one chain to another. This enables the workload distribution to multiple subsets of nodes that run the same protocol.

There are several proposals that, like ByzCoin, target the consensus mechanism and try to improve different aspects. Ripple [52] implements and runs a variant of PBFT that is low-latency and based on collectively-trusted subnetworks with slow membership changes. The degree of decentralization depends on the concrete configuration of an instance. Tendermint [10] also implements a PBFT-like algorithm, but evaluates it with at most 64 “validators”. Furthermore, Tendermint does not address important challenges such as the link-bandwidth between validators, which we found to be a primary bottleneck. PeerCensus [19] is an interesting alternative that shares similarities with ByzCoin, but is only a preliminary theoretical analysis.

Finally, Stellar [43] proposes a novel consensus protocol named Federated Byzantine Agreement, which introduces Quorum slices that enable a BFT protocol “open for anyone to participate”. Its security, however, depends on a nontrivial and unfamiliar trust model requiring correct configuration of trustees by each client.

7 Limitations and Future Work

This section briefly outlines several of ByzCoin’s important remaining limitations, and areas for future work.

Consensus-Group Exclusion. A malicious ByzCoin leader can potentially exclude nodes from the consensus process. This is easier in the flat variant, where the leader is responsible for contacting every participating miner, but it is also possible in the tree-based version, if the leader “reorganizes” the tree and puts nodes targeted for exclusion in subtrees where the roots are colluding nodes. A malicious leader faces a dilemma, though: excluded nodes lose their share of newly minted coins which increases the overall value per coin and thus the leader’s reward. The victims, however, will quickly broadcast view-change messages in an attempt to remove the Byzantine leader.

As an additional countermeasure to mitigate such an attack, miners could run a peer-to-peer network on top of the tree to communicate protocol details. Thus each node potentially receives information from multiple sources. If the parent of a node fails to deliver the announcement message of a new round, this node could then choose to attach itself (together with its entire subtree) to another participating (honest) miner. This self-adapting

tree could mitigate the leader’s effort to exclude miners. As a last resort, the malicious leader could exclude the commitments of the victims from the aggregate commitment, but as parts of the tree have witnessed these commitments, the risk of triggering a view-change is high.

In summary, the above attack seems irrational as the drawbacks of trying to exclude miners seem to outweigh the benefits. We leave a more thorough analysis of this situation for future work.

Defenses Against 33%+ Attacks. An attacker powerful enough to control more than $\frac{1}{3}$ of the consensus shares can, in the Byzantine threat model, trivially censor transactions by withholding votes, and double-spend by splitting honest nodes in two disjoint groups and collecting enough signatures for two conflicting microblocks. Fig. 12 shows how the safety of ByzCoin fails at 30%, whereas Bitcoin remains safe even for 48%, if a client can wait long enough.

However, the assumption that an attacker completely controls the network is rather unrealistic, especially if messages are authenticated and spoofing is impossible [3]. The existence of the peer-to-peer network on top of the tree, mentioned in the previous paragraph, enables the detection of equivocation attacks such as microblock forks and mitigates the double-spending efforts, as honest nodes will stop following the leader. Thus, double-spending and history rewriting attacks in ByzCoin become trivial only after the attacker has 66% of the shares, effectively increasing the threshold from 51% to 66%. This assumption is realistic, as an attacker controlling the complete network can actually split Bitcoin’s network in two halves and trivially double-spend on the weaker side. This is possible because the weak side creates blocks that will be orphaned once the partition heals. We again leave a more thorough analysis of this situation for future work.

Proof-of-Work Alternatives. Bitcoin’s hash-based proof-of-work has many drawbacks, such as energy waste and the efficiency advantages of custom ASICs that have made mining by “normal users” impractical. Many promising alternatives are available, such as memory-intensive puzzles [4], or proof-of-stake designs [37]. Consensus group membership might in principle also be based on other Sybil attack-resistant methods, such as those based on social trust networks [58]. A more democratic alternative might be to apportion mining power on a “1 person, 1 vote” principle, based on anonymous *proof-of-personhood* tokens distributed at pseudonym parties [28]. Regardless, we treat the ideal choice of Sybil attack-resistance mechanism as an issue for future work, orthogonal to the focus of this paper.

Other Directions. Besides the issues outlined above, there are many more interesting open questions worth considering: Sharding [17] presents a promising approach to scale distributed protocols and was already studied for private blockchains [18]. A sharded variant of ByzCoin might thus achieve even better scalability and performance numbers. A key obstacle that needs to be analyzed in that context before though is the generation of bias-resistant public randomness [40] which would enable to pick members of a shard in a distributed and secure manner. Yet another challenge is to find ways to increase incentives of rational miners to remain honest, like binding coins and destroying them when misbehavior is detected [10]. Finally, asynchronous BFT [12, 11] is another interesting class of protocols, which only recently started to be analyzed in the context of blockchains [46].

8 Conclusion

ByzCoin is a scalable Byzantine fault tolerant consensus algorithm for open decentralized blockchain systems such as Bitcoin. ByzCoin’s strong consistency increases Bitcoin’s core security guarantees—shielding against attacks on the consensus and mining system such as N-confirmation double-spending, intentional blockchain forks, and selfish mining—and also enables high scalability and low transaction latency. ByzCoin’s application to Bitcoin is just one example, though: theoretically, it can be deployed to any blockchain-based system, and the proof-of-work-based leader election mechanism might be changed to another approach such as proof-of-stake. If open membership is not an objective, the consensus group could be static, though still potentially large. We developed a wide-scale prototype implementation of ByzCoin, validated its efficiency with measurements and experiments, and have shown that Bitcoin can increase the capacity of transactions it handles by more than two orders of magnitude.

Acknowledgments

We would like to thank the DeterLab project team for providing the infrastructure for our experimental evaluation, Joseph Bonneau for his input on our preliminary design, and the anonymous reviewers for their helpful feedback.

References

- [1] ABD-EL-MALEK, M., GANGER, G. R., GOODSON, G. R., REITER, M. K., AND WYLIE, J. J. Fault-scalable Byzantine Fault-tolerant Services.
- [2] ANDRESEN, G. Classic? Unlimited? XT? Core?, Jan. 2016.
- [3] APOSTOLAKI, M., ZOHAR, A., AND VAN-BEVER, L. Hijacking Bitcoin: Large-scale Network Attacks on Cryptocurrencies. *arXiv preprint arXiv:1605.07524* (2016).
- [4] ATENIESE, G., BONACINA, I., FAONIO, A., AND GALESI, N. Proofs of Space: When Space is of the Essence. In *Security and Cryptography for Networks*. Springer, 2014, pp. 538–557.
- [5] BACK, A., CORALLO, M., DASHJR, L., FRIEDENBACH, M., MAXWELL, G., MILLER, A., POELSTRA, A., TIMÓN, J., AND WUILLE, P. Enabling Blockchain Innovations with Pegged Sidechains.
- [6] BERNSTEIN, D. J., DUIF, N., LANGE, T., SCHWABE, P., AND YANG, B.-Y. High-speed high-security signatures. *Journal of Cryptographic Engineering* 2, 2 (2012), 77–89.
- [7] BITCOIN WIKI. Confirmation, 2016.
- [8] BITCOIN WIKI. Scalability, 2016.
- [9] BONNEAU, J., MILLER, A., CLARK, J., NARAYANAN, A., KROLL, J., AND FELTEN, E. W. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*. IEEE (2015).
- [10] BUCHMAN, E. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains, 2016.
- [11] CACHIN, C., KURSAWE, K., PETZOLD, F., AND SHOUP, V. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology (CRYPTO)* (Aug. 2001).
- [12] CACHIN, C., KURSAWE, K., AND SHOUP, V. Random Oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *19th ACM Symposium on Principles of Distributed Computing (PODC)* (July 2000).
- [13] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., AND SINGH, A. SplitStream: high-bandwidth multicast in cooperative environments. In *ACM Symposium on Operating Systems Principles (SOSP)* (2003).

- [14] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Feb. 1999).
- [15] CLEMENT, A., WONG, E. L., ALVISI, L., DAHLIN, M., AND MARCHETTI, M. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *6th USENIX Symposium on Networked Systems Design and Implementation* (Apr. 2009).
- [16] COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., AND SHRIRA, L. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 177–190.
- [17] CROMAN, K., DECKE, C., EYAL, I., GENCER, A. E., JUELS, A., KOSBA, A., MILLER, A., SAXENA, P., SHI, E., SIRER, E. G., AN, D. S., AND WATTENHOFER, R. On Scaling Decentralized Blockchains (A Position Paper). In *3rd Workshop on Bitcoin and Blockchain Research* (2016).
- [18] DANEZIS, G., AND MEIKLEJOHN, S. Centrally Banked Cryptocurrencies.
- [19] DECKER, C., SEIDEL, J., AND WATTENHOFER, R. Bitcoin Meets Strong Consistency. In *17th International Conference on Distributed Computing and Networking (ICDCN)*, Singapore (January 2016).
- [20] DECKER, C., AND WATTENHOFER, R. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Stabilization, Safety, and Security of Distributed Systems*. Springer, Aug. 2015, pp. 3–18.
- [21] DEERING, S. E., AND CHERITON, D. R. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems* 8, 2 (May 1990).
- [22] DeterLab Network Security Testbed, September 2012.
- [23] DOUCEUR, J. R. The Sybil Attack. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)* (Mar. 2002).
- [24] EYAL, I., GENCER, A. E., SIRER, E. G., AND VAN RENESSE, R. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation* (NSDI 16) (Santa Clara, CA, Mar. 2016), USENIX Association.
- [25] EYAL, I., AND SIRER, E. G. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*. Springer, 2014, pp. 436–454.
- [26] FINNEY, H. Best practice for fast transaction acceptance – how high is the risk?, Feb. 2011. Bitcoin Forum comment.
- [27] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.
- [28] FORD, B., AND STRAUSS, J. An offline foundation for online accountable pseudonyms. In *1st International Workshop on Social Network Systems (SocialNets)* (2008).
- [29] GARAY, J., KIAYIAS, A., AND LEONARDOS, N. The Bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT 2015*. Springer, 2015, pp. 281–310.
- [30] GERVAIS, A., KARAME, G. O., WUST, K., GLYKANTZIS, V., RITZDORF, H., AND CAPKUN, S. On the Security and Performance of Proof of Work Blockchains. Tech. rep., IACR: Cryptology ePrint Archive, 2016.
- [31] GERVAIS, A., RITZDORF, H., KARAME, G. O., AND CAPKUN, S. Tampering with the Delivery of Blocks and Transactions in Bitcoin. In *22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 692–705.
- [32] GUERRAOUI, R., KNEŽEVIĆ, N., QUÉMA, V., AND VUKOLIĆ, M. The next 700 BFT protocols. In *5th European conference on Computer systems* (2010), ACM, pp. 363–376.
- [33] HEARN, M., AND SPILMAN, J. Rapidly-adjusted (micro)payments to a pre-determined party, 2015.
- [34] HEILMAN, E., KENDLER, A., ZOHAR, A., AND GOLDBERG, S. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *24th USENIX Security Symposium* (2015), pp. 129–144.
- [35] KARAME, G. O., ANDROULAKI, E., AND CAPKUN, S. Double-spending fast payments in Bitcoin. In *19th ACM Conference on Computer and communications security* (2012), ACM, pp. 906–917.

- [36] KIAYIAS, A., AND PANAGIOTAKOS, G. Speed-Security Tradeoffs in Blockchain Protocols. Tech. rep., IACR: Cryptology ePrint Archive, 2015.
- [37] KING, S., AND NADAL, S. PPCoin: Peer-to-peer Crypto-Currency with Proof-of-Stake.
- [38] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. In *21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (Oct. 2007), ACM.
- [39] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401.
- [40] LENSTRA, A. K., AND WESOLOWSKI, B. A random zoo: sloth, unicorn, and trx. IACR eprint archive, Apr. 2015.
- [41] LEWENBERG, Y., SOMPOLINSKY, Y., AND ZOHAR, A. Inclusive Block Chain Protocols. In *Financial Cryptography and Data Security*. Springer, Jan. 2015, pp. 528–547.
- [42] LINUX FOUNDATION. Hyperledger Project, 2016.
- [43] MAZIÈRES, D. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus.
- [44] MERKLE, R. C. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, June 1979.
- [45] MILLER, A., AND JANSEN, R. Shadow-Bitcoin: scalable simulation via direct execution of multi-threaded applications. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)* (2015).
- [46] MILLER, A., XIA, Y., CROMAN, K., SHI, E., AND SONG, D. The honey badger of BFT protocols. Tech. rep., Cryptology ePrint Archive 2016/199, 2016.
- [47] NAKAMOTO, S. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.
- [48] NAYAK, K., KUMAR, S., MILLER, A., AND SHI, E. Stubborn Mining: Generalizing Selfish Mining and Combining with an Eclipse Attack. In *1st IEEE European Symposium on Security and Privacy* (Mar. 2015).
- [49] PEASE, M., SHOSTAK, R., AND LAMPORT, L. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)* 27, 2 (1980), 228–234.
- [50] POON, J., AND DRYJA, T. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, Jan. 2016.
- [51] SCHNORR, C. P. Efficient signature generation by smart cards. *Journal of Cryptology* 4, 3 (1991), 161–174.
- [52] SCHWARTZ, D., YOUNGS, N., AND BRITTO, A. The Ripple protocol consensus algorithm. *Ripple Labs Inc White Paper* (2014), 5.
- [53] SOMPOLINSKY, Y., AND ZOHAR, A. Accelerating Bitcoin’s Transaction Processing. Fast Money Grows on Trees, Not Chains, Dec. 2013.
- [54] SYTA, E., TAMAS, I., VISHER, D., WOLINSKY, D. I., L., GAILLY, N., KHOFFI, I., AND FORD, B. Keeping Authorities “Honest or Bust” with Decentralized Witness Cosigning. In *37th IEEE Symposium on Security and Privacy* (May 2016).
- [55] VENKATARAMAN, V., YOSHIDA, K., AND FRANCIS, P. Chunkspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast. In *14th International Conference on Network Protocols (ICNP)* (Nov. 2006).
- [56] VUKOLIĆ, M. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International Workshop on Open Problems in Network Security* (2015), Springer, pp. 112–125.
- [57] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).
- [58] YU, H., GIBBONS, P. B., KAMINSKY, M., AND XIAO, F. SybilLimit: A Near-Optimal Social Network Defense against Sybil Attacks. In *29th IEEE Symposium on Security and Privacy (S&P)* (May 2008).

Faster Malicious 2-party Secure Computation with Online/Offline Dual Execution

Peter Rindal*

Oregon State University

Mike Rosulek*

Oregon State University

Abstract

We describe a highly optimized protocol for general-purpose secure two-party computation (2PC) in the presence of malicious adversaries. Our starting point is a protocol of Kolesnikov et al. (TCC 2015). We adapt that protocol to the online/offline setting, where two parties repeatedly evaluate the same function (on possibly different inputs each time) and perform as much of the computation as possible in an offline preprocessing phase before their inputs are known. Along the way we develop several significant simplifications and optimizations to the protocol.

We have implemented a prototype of our protocol and report on its performance. When two parties on Amazon servers in the same region use our implementation to securely evaluate the AES circuit 1024 times, the amortized cost per evaluation is *5.1ms offline + 1.3ms online*. The total offline+online cost of our protocol is in fact less than the *online* cost of any reported protocol with malicious security. For comparison, our protocol’s closest competitor (Lindell & Riva, CCS 2015) uses 74ms offline + 7ms online in an identical setup.

Our protocol can be further tuned to trade performance for leakage. As an example, the performance in the above scenario improves to *2.4ms offline + 1.0ms online* if we allow an adversary to learn a single bit about the honest party’s input with probability 2^{-20} (but not violate any other security property, e.g. correctness).

1 Introduction

Secure two-party computation (2PC) allows mutually distrusting parties to perform a computation on their combined inputs, while revealing only the result. 2PC was conceived in a seminal paper by Yao [34] and shown to be feasible in principle using a construction now known as *garbled circuits*. Later, the Fairplay

project [24] was the first implementation of Yao’s protocol, which inspired interest in the practical performance of 2PC.

1.1 Cut & Choose, Online/Offline Setting

The leading technique to secure Yao’s protocol against malicious adversaries is known as *cut-and-choose*. The idea is to have the sender generate many garbled circuits. The receiver will choose a random subset of these to be checked for correctness. If all checked circuits are found to be correct, then the receiver has some confidence about the unopened circuits, which can be evaluated.

The cost of the cut-and-choose technique is therefore tied to the number of garbled circuits that are generated. To restrict a malicious adversary to a 2^{-s} chance of violating security, initial cut-and-choose mechanisms required approximately 17s circuits [20]. This overhead was later reduced to 3s circuits [21, 31, 32] and then s circuits [19].

Suppose two parties wish to perform N secure computations of the same function f (on possibly different inputs each time), and are willing to do offline preprocessing (which does not depend on the inputs). In this *online/offline setting*, far fewer garbled circuits are needed per execution. The idea, due to [14, 22], is to generate many garbled circuits (enough for all N executions) and perform a single cut-and-choose on them all. Then each execution of f will evaluate a *random* subset (typically called a *bucket*) of the unopened circuits. Because the unopened circuits are randomly assigned to executions, only $O(s/\log N)$ circuits are needed per bucket to achieve security 2^{-s} . Concretely, 4 circuits per bucket suffice for security 2^{-40} and $N = 1024$.

1.2 Dual-execution Paradigm

An alternative to cut-and-choose for malicious-secure 2PC is the dual-execution protocol of Mohassel & Franklin [25], which requires only two garbled circuits.

*Supported by NSF award 1149647. The first author is also supported by an ARCS foundation fellowship.

The idea is that two parties run two instances of Yao’s protocol, with each party acting as sender in one instance and receiver in the other. They then perform a *reconciliation* step in which their garbled outputs are securely compared for equality. Intuitively, one of the garbled outputs is guaranteed to be correct, so the reconciliation step allows the honest party to check whether its garbled output agrees with the correct one held by the adversary.

Unfortunately, the dual execution protocol allows an adversary to learn an arbitrary bit about the honest party’s input. Consider an adversary who instead of garbling the function f , garbles a different function f' . Then the output of the reconciliation step (secure equality test) reveals whether $f(x_1, x_2) = f'(x_1, x_2)$. However, it can be shown that the adversary can learn *only* a single bit, and, importantly, cannot violate output *correctness* for the honest party.

1.3 Reducing Leakage in Dual-execution

Kolesnikov et al. [16] proposed a combination of dual-execution and cut-and-choose that reduces the probability of a leaked bit. The idea is for each party to garble and send s circuits instead of 1, and perform a cut-and-choose to check each circuit with probability $1/2$. Each circuit should have the same garbled encoding for its outputs, so if both parties are honest, both should receive just one candidate output.

However, a malicious party could cause the honest party to obtain several candidate outputs. The approach taken in [16] is to have the parties use *private set intersection (PSI)* to find a common value among their *sets* of reconciliation values. This allows the honest party to identify which of its candidate outputs is the correct one.

In Section 4 we discuss in more detail the security offered by this protocol. Briefly, an adversary cannot violate output correctness for the honest party, and learns only a *single bit* about the honest party’s input with probability at most $1/2^s$ (which happens only when the honest party doesn’t evaluate any correct garbled circuit).

2 Overview of Our Results

We adapt the dual-execution protocol of [16] to the online/offline setting. The result is the fastest protocol to date for 2PC in the presence of malicious adversaries. At a very high level, both parties exchange many garbled circuits in the offline phase and perform a cut-and-choose. In the online phase, each party evaluates a random bucket of its counterpart’s circuits. The parties then use the PSI-based reconciliation to check the outputs.

2.1 Technical Contributions

While the high-level idea is straight-forward, some non-trivial technical changes are necessary to adapt [16] to the online/offline setting while ensuring high performance in practice.

In particular, an important part of any malicious-secure protocol is to ensure that parties use the same inputs in all garbled circuits. The method suggested in [16] is incompatible with offline pre-processing, whereas the method from [23] does not ensure consistency between circuits generated by different parties, which is the case for dual-execution (both parties generate garbled circuits). We develop a new method for input consistency that is tailored specifically to the dual-execution paradigm and that incurs less overhead than any existing technique.

In [16], the parties evaluate garbled circuits and then use *active-secure* private set intersection (PSI) to reconcile their outputs. We improve the analysis of [16] and show that it suffices to use PSI that gives a somewhat weaker level of security. Taking advantage of this, we describe an extremely lightweight PSI protocol (a variant of one in [30]) that satisfies this weak level of security while being round-optimal.

2.2 Implementation, Performance

We implemented a C++ prototype of our protocol using state-of-the-art optimizations, including the garbled-circuit construction of [35]; the OT-extension protocol of [15] instantiated with the base OTs of [7]. The prototype is heavily parallelized within both phases. Work is divided amongst threads that concurrently generate & evaluate circuits, allowing network throughput to be the primary bottleneck. The result is an extremely fast 2PC system. When securely evaluating the AES circuit on co-located Amazon AWS instances, we achieve the lowest amortized cost to date of 5.1ms offline + 1.3ms online per execution.

2.3 Comparison to GC-based Protocols

There have been several implementations of garbled-circuit-based 2PC protocols that achieve malicious security [1, 12, 18, 23, 29, 31, 32]. Except for [23], none of these protocols are in the online/offline settings so their performance is naturally much lower ($100\text{-}1000\times$ slower than online/offline protocols). Among them, the fastest reported secure evaluation of AES is that of [12], which was 0.46s exploiting consumer GPUs. Other protocols have been described (but not implemented) that combine cut-and-choose with the dual-execution paradigm to achieve malicious security [13, 26]. The protocol of [13] leaks more than one bit when the adversary successfully cheats during cut-and-choose.

Our protocol is most closely related to that of [23], which also achieves fast, active-secure 2PC in the online/offline setting. [23] is an implementation of the protocol of [22], and we refer to the protocol and its implementation as “LR” in this section. Both the LR protocol and ours are based on garbled circuits but use fundamentally different approaches to achieving malicious secu-

	Input Labels	Reconciliation
LR [23]	$ x (B+B')\kappa_c$	$ x B'\kappa_c$
Us (Async PSI)	$ x B\kappa_c$	$B^2\kappa_s\kappa_c$
Us (Sync PSI)		$B\kappa_s+B^2\kappa_c$

Figure 1: Asymptotic communication costs of the LR protocol vs. ours (comparing online phases). B is the number of circuits in a bucket; $B' \approx 3B$ is the number of auxiliary cheating-recovery circuits in [23]; $|x|$ is length of sender’s inputs; κ_s is the statistical security parameter; κ_c is the computational security parameter.

riety. For clarity, we now provide a list of major differences between the two protocols.

(1) LR uses a more traditional cut-and-choose mechanism where one party acts as sender and the other as receiver & evaluator. Our protocol on the other hand uses a dual-execution paradigm in which both parties play symmetric roles, so their costs are identical.

Since parties act as both sender and receiver, each party performs more work than in the traditional cut-and-choose paradigm. However, the symmetry of dual-execution means that both parties are performing computational work simultaneously, rather than idle waiting. The increase in combined work does not significantly affect *latency* or *throughput* if the communication channel is full-duplex.

(2) Our protocol can provide more flexible security guarantees; in particular, it may be used with smaller parameter choices. In more detail, let κ_s denote a statistical security parameter, meaning that the protocol allows the adversary to completely break security with probability $1/2^{\kappa_s}$. In the LR protocol, a failure of the cut-and-choose step can violate all security properties, so the number of garbled circuits is proportional to κ_s .

Our protocol has an additional parameter κ_b , where the protocol leaks (only) a single bit to the adversary with probability $1/2^{\kappa_b}$. In our protocol (as in [16]), the number of garbled circuits is proportional to κ_b . When instantiated with $\kappa_b = \kappa_s = 40$, our protocol gives an equivalent guarantee to the LR protocol with $\kappa_s = 40$. From this baseline, our protocol allows either κ_s to be increased (strictly improving the security guarantee without involving more garbled circuits) or κ_b to be decreased (trading performance for a small chance of a single bit leaking).¹

(3) Our online phase has superior asymptotic cost, stemming from the differences in protocol paradigm — see a summary in Figure 1. LR uses a *cheating-recovery phase*, introduced in [19]: after evaluating the main circuits, the parties evaluate auxiliary circuits that allow the receiver to learn the sender’s input if the receiver can

¹For example, two parties might want to securely evaluate AES a million times on the same secret-shared key each time, where the key is not used for anything else. In that case, a $1/2^{20}$ or $1/2^{30}$ chance of leaking a single bit about this key might be permissible.

“prove” that the sender was cheating. Our protocol uses the PSI-based dual-execution reconciliation phase.

The important difference is that in the LR protocol, the sender’s input is provided to both the main circuits and auxiliary circuits. If there are B main garbled circuits in a bucket, then there are $B' \approx 3B$ auxiliary circuits, and garbled inputs must be sent for all of them in the online phase. Each individual garbled input is sent by decommitting to an offline commitment, so it contributes to communication as well as a call to a hash function. Furthermore, the cheating-recovery phase involves decommitments to garbled outputs for the auxiliary circuits, which are again proportional to the sender’s input length.

In contrast, our protocol uses no auxiliary circuits so has less garbled inputs to send (and less associated decommitments to check). Our reconciliation phase scales only with B and is independent of the parties’ input size. The overall effect is that our online phase involves significantly less communication and computation, with the difference growing as the computations involve longer inputs. With typical parameters $B = 4$ and $\kappa_s = 40$, our reconciliation phase is cheaper whenever $|x| \geq 54$ bits. Even for the relatively small AES circuit, our protocol sends roughly $10\times$ less data in the online phase.

(4) LR’s online phase uses 4 rounds of interaction² and delivers output only to one party. If both parties require output, their protocol must be modified to include an additional round. Our online phase also delivers outputs to both parties using either 5 or 6 rounds (depending on the choice of PSI subprotocols). We conjecture that our protocol can be modified to use only 4 rounds, but leave that question to follow-up work.

(5) Our implementation is more efficient than LR. The offline phase more effectively exploits parallelism and LR is implemented using a mix of Java & C++. The architecture of LR has a serial control flow with computationally heavy tasks performed in parallel using low level C++ code. In contrast, our protocol implementation is in C++ and fully parallelized with low level synchronization primitives.

2.4 Comparison to Non-GC Protocols

Another paradigm for malicious security in the online/offline setting is based not on garbled circuits but arithmetic circuits and secret sharing. Notable protocols and implementations falling into this paradigm include [8, 9, 10, 11, 27]. These protocols indeed have

²For our purposes, a *round* refers to both parties sending a message. In other words, messages in the same round are allowed to be sent simultaneously, and our implementation takes advantage of full-duplex communication to reduce latency. We emphasize that synchronicity is *not* required for our security analysis. The protocol is secure against an adversary who waits to obtain the honest party’s message in round i before sending its own round i message.

lightweight online phases, and many instances can be batched in parallel to achieve *throughput* comparable to our protocol. However, all of these protocols have an online phase whose number of rounds depends on the depth of the circuit being evaluated. As a result, they suffer from significantly higher *latency* than the constant-round protocols in the garbled circuit paradigm like ours. The latest implementations of [9] can securely evaluate AES with online latency 20ms [33]. Of special note is the implementation of the [11] protocol reported in [10], which achieves latency of only 6ms to evaluate AES. However, the implementation is heavily optimized for the special case of computing AES, and it is not clear how applicable their techniques are for general-purpose MPC. In any case, no protocol has reported online latency for AES that is less than our protocol’s total offline+online cost.

The above protocols based on secret-sharing also have significantly more expensive offline phases. Not all implementations report the cost of their offline phases, but the latest implementations of the [9] protocol require 156 seconds of offline time for securely computing AES [33]; many orders of magnitude more than ours. We note that the protocols in the secret-sharing paradigm have an offline phase which does *not* depend on the function that will be evaluated, whereas ours does.

3 Preliminaries

Secure computation. We use the standard notion of universally composable (UC) security [6] for 2-party computation. Briefly, the protocol is secure if for every adversary attacking the protocol, there is a straight-line simulator attacking the ideal functionality that achieves the same effect. We assume the reader is familiar with the details.

We define the ideal functionality $\mathcal{F}_{\text{multi-sfe}}$ that we achieve in Figure 2. The functionality allows parties to evaluate the function f , N times. Adversaries have the power to delay (perhaps indefinitely) the honest party’s output, which is typical in the setting of malicious security. In other words, the functionality does not provide output fairness.

Furthermore, the functionality occasionally allows the adversary to learn an arbitrary additional bit about the inputs. This leakage happens according to the distribution \mathcal{L} chosen by the adversary at setup time. The probability of a leaked bit in any *particular* evaluation of f is guaranteed to be at most ϵ . Further, the leakage is “risky” in the sense that the honest party detects cheating when the leaked bit is zero.

Building blocks. In Figures 10 & 11 we define oblivious transfer (OT) and commitment functionalities that are used in the protocol. In the random oracle model, where H is the random oracle, a party can commit to v by choosing random $r \leftarrow \{0, 1\}^{k_c}$ and sending $c = H(r \| v)$.

We use and adapt the Garbled Circuit notation and terminology of [5]; for a formal treatment, consult that paper. In Appendix A we define the syntax and security requirements, highlighting the differences we adopt compared to [5].

4 The Dual Execution Paradigm

We now give a high-level outline of the (non-online/offline) 2PC protocol paradigm of [16], which is the starting point for our protocol. The protocol makes use of a **two-phase PSI subprotocol**. In the first phase, both parties become committed to their PSI inputs; in the second phase, the PSI output is revealed. This component is modeled in terms of the $\mathcal{F}_{\text{psi}}^{n, \ell}$ functionality in Figure 12.

Assume the parties agree on a function f to be evaluated on their inputs. The protocol is symmetric with respect to Alice and Bob, and for simplicity we describe only Alice’s behavior.

- (1) Alice generates κ_b garbled circuits computing f , using a common garbled output encoding for all of them.
- (2) Alice announces a random subset of Bob’s circuits to open. However, the actual checking of the circuits is delayed until later in the protocol.
- (3) Alice uses OT to receive garbled inputs for the circuits generated by Bob, as in Yao’s protocol. Alice sends the garbled circuits she generated, along with her own garbled input for these circuits.
- (4) Alice evaluates the garbled circuits received from Bob. If Bob is honest, then all of his circuits use the same garbled output encoding and Alice will receive the same garbled output from each one. But in the general case, Alice might obtain several inconsistent garbled outputs.
- (5) Assume that Alice can decode the garbled outputs to obtain the logical circuit output. For each candidate circuit output y with garbled encoding Y_y^b (b for a garbled output under Bob’s encoding), let Y_y^a denote the encoding of y under Alice’s garbled output encoding (which Alice can compute). Interpreting Y_y^a and Y_y^b as *sets* of individual wire labels, let R_y be the XOR of all items in $Y_y^a \cup Y_y^b$, which we write as $R_y = \bigoplus [Y_y^a \cup Y_y^b]$ and which we call the *reconciliation value* for y . Alice sends the set of all $\{R_y\}$ values as input to a PSI instance.
- (6) With the PSI inputs committed, the parties open and check the circuits chosen in the cut-and-choose step. They abort if any circuit is not correctly garbled, or the circuits do not have consistent garbled output encodings.
- (7) The parties release the PSI output. Alice aborts if the PSI output is not a singleton set. Otherwise, if

Setup stage: On common input $(\text{SETUP}, f, N, \varepsilon)$ from both parties, where f is a boolean circuit:

- If neither party is corrupt, set $L = 0^N$. Otherwise, wait for input $(\text{CHEAT}, \mathcal{L})$ where \mathcal{L} is a distribution over $\{0, 1\}^N \cup \{\perp\}$ with the property that for every i , $\Pr_{L \leftarrow \mathcal{L}}[L_i = 1] \leq \varepsilon$. Sample $L \leftarrow \mathcal{L}$ using random coins χ and give $(\text{CHEATRESULT}, \chi)$ to the adversary. If $L = \perp$ then give output (CHEATING!) to the honest party and stop responding.
- Send output (READY) to both parties. Initialize counter $ctr = 1$. Proceed to the execution stage.

Execution stage: Upon receiving inputs (INPUT, x_1) from P_1 and (INPUT, x_2) from P_2 :

- Compute $z = f(x_1, x_2)$. If both parties are honest, give (OUTPUT, ctr, z) to both parties.
- If any party is corrupt, give (OUTPUT, ctr, z) to the adversary.
- If $L_{ctr} = 1$, wait for a command (LEAK, P) from the adversary, where P is a boolean predicate. Compute $p = P(x_1, x_2)$ and give $(\text{LEAKRESULT}, p)$ to the adversary.
- If any party is corrupt, then on input (DELIVER) from the adversary, if $p = 0$ above, then give output (CHEATING!) to the honest party, else give output (OUTPUT, ctr, z) to the honest party.
- If $ctr = N$ then stop responding; otherwise set $ctr = ctr + 1$ and repeat the execution stage.

Figure 2: The $(\varepsilon\text{-leaking})$ secure function evaluation functionality $\mathcal{F}_{\text{multi-sfe}}$.

the output is $\{R^*\}$ then Alice outputs the value y such that $R^* = R_y$.

4.1 Security Analysis and Other Details

Suppose Alice is corrupt and Bob is honest. We will argue that Alice learns nothing beyond the function output, except that with probability $2^{-\kappa_b}$ she learns a single bit about Bob’s input.

Suppose Alice uses input x_1 as input to the OTs, and Bob has input x_2 . Since Bob’s circuits are honestly generated and use the same garbled output encoding, every circuit evaluated by Alice leads to the same garbled output $Y_{y^*}^b$ that encodes logical value $y^* = f(x_1, x_2)$. Note that by the *authenticity* property of the garbled circuits, this is the *only* valid garbled output that Alice can predict.

Since Alice may generate malicious garbled circuits, honest Bob may obtain several candidate outputs from these circuits. Bob’s input to the PSI computation will be a collection of reconciliation values, each of the form $R_y = \bigoplus [Y_y^a \cup Y_y^b]$.

At the time of PSI input, none of Bob’s (honestly) garbled circuits have been opened, so they retain their authenticity property. Then Alice cannot predict any *valid* reconciliation value except for this R_{y^*} . This implies that the PSI output will be either $\{R_{y^*}\}$ or \emptyset . In particular, Bob will either abort or output the correct output y^* . Furthermore, the output of the PSI computation can be simulated knowing only whether honest Bob has included R_{y^*} in his PSI input.

The protocol includes a mechanism to ensure that Alice uses the same x_1 input for all of the garbled circuits. Hence, if Bob evaluates *at least one* correctly generated garbled circuit, it will give output y^* and Bob will surely include the R_{y^*} reconciliation value in his PSI input. In that case, the PSI output can be simulated as usual.

The probability the Alice manages to make Bob evaluate *no* correctly generated garbled circuits is $2^{-\kappa_b}$ — she would have to completely predict Bob’s cut-and-choose challenge to make all opened circuits correct and all evaluated circuits incorrect. But even in this event, the simulator only needs to know whether $f'(x_1, x_2) = y^*$ for any of the f' computed by Alice’s malicious garbled circuits. This is only one bit of information about x_2 which the simulator can request from the ideal functionality.

4.2 Outline for Online/Offline Dual-Execution

Our high-level approach is to adapt the [16] protocol to the online/offline setting. The idea is that the two parties plan to securely evaluate the same function f , N times, on possibly different inputs each time. In preparation they perform an offline pre-processing phase that depends only on f and N , but not on the inputs. They generate many garbled circuits and perform a cut-and-choose on all of them. Then the remaining circuits are assigned randomly to *buckets*. Later, once inputs are known in the online phase, one bucket’s worth of garbled circuits are consumed for each evaluation of f .

Our protocol will leak a single bit about the honest party’s input only when a bucket contains no “good” circuit from the adversary (where “good” is the condition that is verified for opened circuits during cut-and-choose). Following the lead of [23], we focus on choosing the number of circuits so that the probability of such an event in *any particular* bucket is $2^{-\kappa_b}$. We note that the analysis of parameters in [14, 22] considers an *overall* cheating condition, i.e., that *there exists* a bucket that has no “good” circuits, which leads to slightly different numbers.

Lemma 1 ([23]). *If the parties plan to perform N exe-*

cutions, using a bucket of B circuits for each execution and a total of $\hat{N} \geq NB$ garbled circuits generated for the overall cut-and-choose, then the probability that a specific bucket contains no good circuit is at most:

$$\max_{t \in \{B, \dots, NB\}} \left\{ \frac{\binom{\hat{N}-t}{NB-t}}{\binom{\hat{N}}{NB}} \cdot \frac{\binom{t}{B}}{\binom{NB}{B}} \right\}.$$

Suppose the parties will perform N executions, using buckets of size B in the online phase, and wish for 2^{-k_b} probability of leakage. We can use the formula to determine the smallest compatible \hat{N} . In the full version we show all reasonable parameter settings for $k_b \in \{20, 40, 80\}$ and $N \in \{8, 16, 32, \dots, 32768\}$.

By adapting [16] to the online/offline setting, we obtain the generic protocol outlined in Figure 3. Even with pre-processing, an online OT requires two rounds, one of which can be combined with the direct sending of garbled inputs. The protocol therefore requires three rounds plus the number of rounds needed for the PSI subprotocol (at least two).

4.3 Technicalities

We highlight which parts of the [16] protocol break down in the online/offline setting and require technical modification:

Same garbled output encoding. In [16] each party is required to generate garbled circuits that have a common output encoding. Their protocol includes a mechanism to enforce this property. In our setting, we require each bucket of circuits to have the same garbled output encoding. But this is problematic because in our setting a garbled circuit is generated before the parties know which bucket it will be assigned to.

Our solution is to have the garbler provide for each bucket a *translation* of the following form. The garbler chooses a bucket-wide garbled output encoding; e.g., for the first output wire, he chooses wire labels W_0^*, W_1^* encoding false and true, respectively. Then if W_0^j, W_1^j are the output wire labels already chosen for the j th circuit in this bucket, the garbler is supposed to provide *translation values* $W_v^j \oplus W_v^*$ for $v \in \{0, 1\}$. After evaluating, the receiver will use these values to translate the garbled input to this bucket-wide encoding that is used for PSI reconciliation.

Of course, a cheating party can provide invalid translation values. So we use step 3 of the online phase (Figure 3) to check them. In more detail, a sender must commit in the offline phase to the output wire labels of every garbled circuit. These will be checked if the circuit is chosen in the cut-and-choose. In step 3 of the online phase, these commitments are opened so that the receiver can check the consistency of the translation values (i.e., whether

Offline phase:

- (1) Parties perform offline preprocessing for the OTs that will be needed, and for the PSI subprotocol, if appropriate.
- (2) Based on N and κ_b , the parties determine appropriate parameters \hat{N}, B according to the discussion in Section 4.2. Each party generates and sends \hat{N} garbled circuits, and chooses a random subset of $N - NB$ of their counterpart's circuits to be opened. The chosen circuits are opened and parties abort if circuits are found to be generated incorrectly.
- (3) Each party randomly assigns their counterpart's circuits to *buckets* of size B . Each online execution will consume one bucket's worth of circuits.

Online phase:

- (1) Parties exchange garbled inputs: For one's own garbled circuits in the bucket, a party directly sends the appropriate garbled inputs; for the counterpart's garbled circuits, a party uses OT as a receiver to obtain garbled inputs as in Yao's protocol.
- (2) Parties evaluate the garbled circuits and compute the corresponding set of reconciliation values. They commit their sets of reconciliation values as inputs to a PSI computation.
- (3) With the PSI inputs committed, the parties open some checking information (see text in Section 4.3) and abort if it is found to be invalid.
- (4) The parties release the PSI output and abort if the output is \emptyset . Otherwise, they output the plaintext value whose reconciliation value is in the PSI output.

Figure 3: High-level outline of the online/offline, dual-execution protocol paradigm.

they map to a hash of the common bucket-wide encoding provided during bucketing.). This step reveals all of the bucket-wide encoding values, making it now easy for an adversary to compute any reconciliation value. This is why we employ a 2-phase PSI protocol, so that PSI inputs are committed before these translation values are checked.

Adaptive garbling. Standard security definitions for garbled circuits require the evaluator to choose the input before the garbled circuit is given. However, the entire purpose of offline pre-processing is to generate & send the garbled circuits before the inputs are known. This requires the garbling scheme to satisfy an appropriate *adaptive security* property, which is common to all works in the online/offline setting [14, 22]. See Appendix A for details.

Input consistency. To achieve security against active adversaries, GC-based protocols must ensure that parties provide the same inputs to all circuits that are evaluated. This is known as the problem of *input consistency*. The protocol of [16] uses the input consistency mechanism of shelat & Shen [32] which is unfortunately not compatible with the online/offline setting. More details follow in the next section.

5 Input Consistency

In this section we describe a new, extremely lightweight input-consistency technique that is tailored for the dual-execution paradigm.

5.1 Consistency Between Alice’s & Bob’s Circuits

We start with the “classical” dual-execution scenario, where Alice and Bob each generate one garbled circuit. We describe how to force Alice to use the same input in both of these garbled circuits (of course, the symmetric steps are performed for Bob). The high-level idea is to bind her behavior as OT receiver (when obtaining garbled inputs for Bob’s circuits) to the commitments of her garbled inputs in her own circuits.

It is well-known [2] that oblivious transfers on random inputs can be performed offline, and later “derandomized” to OTs of chosen inputs. Suppose two parties perform a random string OT offline, where Alice receives c, m_c and Bob receives m_0, m_1 , for random $c \in \{0, 1\}$ and $m_0, m_1 \in \{0, 1\}^k$. Later when the parties wish to perform an OT of chosen inputs c^* and (m_0^*, m_1^*) , Alice can send $d = c \oplus c^*$ and Bob can reply with $m_0^* \oplus m_d$ and $m_1^* \oplus m_{1 \oplus d}$.

In the offline phase of our protocol, the parties perform a random OT for each Alice-input wire of each circuit, where Alice acts as the receiver. These will be later used for Alice to pick up her garbled input for Bob’s circuit. Let c denote the *string* denoting Alice’s random choice bits for this collection of OTs.

Also in the offline phase, we will have Alice commit to all of the possible garbled input labels for the circuits that she generated. Suppose she commits to them in an order determined by the bits of c ; that is, the wire label commitments for the first input wire are in the order (false,true) if the first bit of c is 0 and (true,false) otherwise.

In the online phase with input x , Alice sends the OT “derandomized” message $d = x \oplus c$. She also sends her garbled inputs for the circuits she generated by opening the commitments indexed by d ; that is, she opens the first or second wire label of the i th pair, depending on whether $d_i = 0$ or $d_i = 1$, respectively. Bob will abort if Alice does not open the correct commitments.

Alice’s effective OT input is $x = d \oplus c$, so she picks

up garbled input corresponding to x . If Alice did indeed commit to her garbled inputs arranged according to c , then she opens the commitments whose *truth values* are also $x = d \oplus c$. More formally,

Offline: Alice garbles the labels A_0, A_1 and Bob garbles B_0, B_1 for Alice’s input. Alice receives OT message m_c and Bob holds m_0, m_1 . Alice sends $(\text{COMMIT}, (\text{sid}, i), A_{i \oplus c})$ to \mathcal{F}_{com} for $i \in \{0, 1\}$.

Online: Alice sends $d = c \oplus x$ to Bob and $(\text{OPEN}, (\text{sid}, d))$ to \mathcal{F}_{com} . Bob receives $(\text{OPEN}, (\text{sid}, d), A_x)$ from \mathcal{F}_{com} and sends $(B_0 \oplus m_d, B_1 \oplus m_{1 \oplus d})$ to Alice who computes $B_x = m_c \oplus (B_{c \oplus d} \oplus m_c)$.

Figure 4: Input consistency on a single bit of Alice’s input for “classic” dual-execution.

Looking ahead, we will use cut-and-choose to guarantee that there is at least one circuit for which Alice’s garbled input commitments are correct in this way.

5.2 Aggregating Several OTs

In our protocol, both parties evaluate a bucket of several circuits. Within the bucket, each of Alice’s circuits is paired with one of Bob’s, as above. However, this implies that Alice uses *separate* OTs to pick up her garbled inputs in each of Bob’s circuits. To address this, we *aggregating* several OTs together to form a single OT.

Suppose Alice & Bob have performed *two* random string OTs, with Alice receiving c, c', m_c, m'_c and Bob receiving m_0, m_1, m'_0, m'_1 , for random $c, c' \in \{0, 1\}$. Suppose further that Alice sends $\delta = c \oplus c'$ to Bob in an offline phase. To aggregate these two random OTs into a *single* chosen-input OT with inputs c^*, m_0^*, m_1^* , Alice can send $d = c \oplus c^*$, and Bob can reply with $m_0^* \oplus (m_d \oplus m'_{d \oplus \delta})$ and $m_1^* \oplus (m_{1 \oplus d} \oplus m'_{1 \oplus d \oplus \delta})$.

The idea extends to aggregate any number B of different random OTs into a single one, with Alice sending $B - 1$ different δ difference values. In our protocol, we aggregate in this way the OTs for the same wire across different circuits. Intuitively, Alice either receives wire labels for the same value on each of these wires (by reporting correct δ values), or else she receives nothing for this wire on *any* circuit.

5.3 Combining Everything with Cut-and-Choose

Now consider a bucket of B circuits. In the offline phase Alice acts as receiver in many random OTs, one collection of them for each of Bob’s circuits. Let c_j be her (string of) choice bits for the OTs associated with the j th circuit. Alice is then supposed to commit to the garbled inputs of her j th circuit arranged according to c_j . Bob will check this property for all circuits that are opened during the cut-and-choose phase by Alice showing the

corresponding OT messages.³ Hence with probability at least $1 - 2^{-\kappa_b}$, at least one circuit in any given bucket has this property. Alice also reports aggregation values $\delta_j = c_1 \oplus c_j$ for these OTs.

In the online phase Alice chooses her input x and sends $d_1 = c_1 \oplus x$ as the OT-derandomization message. This is equivalent to Alice sending $d_j = \delta_j \oplus d_1$ as the message to derandomize the j th OTs. To send her garbled input for the j th circuit, Alice is required to open her commitments indexed by d_j .

If Alice lies in any of the aggregation strings, then she will be missing at least one of the B -out-of- B secret shares which mask her possible inputs. Intuitively, Alice's two strategies are either to provide honest aggregation strings or not obtain any garbled inputs in the position that she lied. In the latter case, the simulator can choose an arbitrary input for Alice in that position.

If we then consider the likely case where Bob's j th circuit is “good” and Alice provided honest aggregation strings, then Alice will have decommitted to inputs for the j th circuit that are consistent with her effective OT input x_1^* . From the discussion in Section 4.1, this is enough to guarantee that the reconciliation phase leaks nothing.

Even if there are no “good” circuits in the bucket (which happens with probability $1/2^{\kappa_b}$), it is still the case that Alice learns no more than if she had received consistent garbled input x_1^* for all of Bob's circuits. So the reconciliation phase can be simulated knowing only whether Bob evaluates any circuit resulting in $f(x_1^*, x_2)$. This is a single bit of information about Bob's input x_2 .

6 Selective Failure Attacks

In the garbled circuit paradigm, suppose Alice is acting as evaluator of some garbled circuits. She uses OT to pick up the wire labels corresponding to her input. A corrupt Bob could provide incorrect inputs to these OTs, so that (for instance) Alice picks up an invalid garbled input if and only if the first bit of her input is 0. By observing whether the evaluator aborts (or produces otherwise unexpected behavior), Bob can deduce the first bit of Alice's input. This kind of attack, where the adversary causes the honest party to abort/fail with probability *depending on its private input* is called a **selective failure attack**.

A common way to prevent selective failure is to use what is called a k -probe-resistant input encoding:

Definition 2 ([20, 32]). *Matrix $M \in \{0, 1\}^{\ell \times n}$ is called **k -probe resistant** if for any $L \subseteq \{1, 2, \dots, n\}$, the Hamming distance of $\bigoplus_{i \in L} M_i$ is at least k , where M_i denotes the i th row vector of M .*

³In fact, since the OT messages are long random strings, Alice can prove that she had particular choice bits in *many* OTs by simply reporting the XOR of all of the corresponding OT messages.

The idea is for Alice to choose a random encoding \tilde{x}_1 of her logical input x_1 satisfying $M\tilde{x}_1 = x_1$. Then the parties evaluate the function $f'(\tilde{x}_1, x_2) = f(M\tilde{x}_1, x_2)$. This additional computation of $M\tilde{x}_1$ involves only XOR operations, so it does not increase the garbled circuit size when using the Free-XOR optimization [17] (it does increase the number OTs needed).

Alice will now use \tilde{x}_1 as her choice bits to the OTs. The adversary can *probe* any number of bits of \tilde{x}_1 , by inserting invalid inputs to the OT in those positions, and seeing whether the other party aborts. For each position probed, the adversary incurs a $1/2$ probability of being caught.⁴

The property of k -probe-resistance implies that probing k bits of the *physical* input \tilde{x}_1 leaks no information about the *logical* input $M\tilde{x}_1$. However, probing k bits incurs a $1 - 2^{-k}$ probability of being caught. Hence, our protocol requires a matrix that is κ_s -probe resistant, where κ_s is the statistical security parameter. We refer the reader to [23] for the construction details of k -probe resistant matrices and their parameters.

6.1 Offlining the k -probe computations

Using k -probe-resistant encodings, the encoded input \tilde{x}_1 is significantly longer than the logical input x_1 . While the computation of $M\tilde{x}_1$ within the garbled circuit can involve no *cryptographic* operations (using Free-XOR), it still involves a quadratic number of XOR operations.

Lindell & Riva [22] suggest a technique that moves these computations associated with k -probe-resistant encodings to the offline phase. The parties will compute the related function $f'(\hat{x}_1, c, x_2) = f(\hat{x}_1 \oplus Mc, x_2)$. In the offline phase, Alice will use OT to obtain wire labels for a random string c . She can also begin to partially evaluate the garbled circuit, computing wire labels for the value Mc .

In the online phase, Alice announces $\hat{x}_1 = x_1 \oplus Mc$ where x_1 is her logical input. Then Bob directly sends the garbled inputs corresponding to \hat{x}_1 . This introduces an asymmetry into our input consistency technique. The most obvious solution to maintain compatibility is to always evaluate circuits of the form $f'(\hat{x}_1, c_1, \hat{x}_2, c_2) = f(\hat{x}_1 \oplus Mc_1, \hat{x}_2 \oplus Mc_2)$, so that Alice uses the same *physical* input (c_1, \hat{x}_1) in both hers and Bob's circuits. However, we would prefer to let Alice use logical input x_1 rather than its (significantly longer) k -probe-encoded input, to reduce the concrete overhead. It turns out that we can accommodate this by exploiting the \mathbb{Z}_2 -linearity of the encoding/decoding operation.

Consider a bucket of circuits $\{1, \dots, B\}$. For the j th

⁴Technically, the sender will commit to all garbled inputs, and then the OTs will be used to transfer the decommitment values. That way, the receiver can abort immediately if an incorrect decommitment value is received.

circuit, Alice acts as receiver in a set of random OTs, and receives random choice bits c_j . The number of OTs per circuit is the number of bits in a k -probe-resistant encoding of Alice’s input.

For Alice’s j th circuit, she must commit to her garbled inputs in the order given by the string Mc_j (rather than just c_j as before). This condition will be checked by Bob in the event that this circuit is opened during cut-and-choose. To assemble a bucket, Alice reports aggregation values $\delta_j = c_1 \oplus c_j$ as before. Imagine Alice derandomizing these OTs by sending an all-zeroes derandomization message. This corresponds to her accepting the random c_1 as her choice bits. (Of course, an all-zeroes message need not be actually sent.) Bob responds and uses the aggregated OTs to send Alice the garbled inputs for c_1 for all of his garbled circuits (indeed, even in the j th circuit Alice receives garbled inputs corresponding to c_1).

In the online phase, Alice decides her logical input x_1 , and she sends $\hat{x}_1 = Mc_1 \oplus x_1$. This value derandomizes the offline k -probe-resistant encoding. Then in her own j th circuit, Alice must open the garbled input commitments indexed by the (public) string $\hat{x}_1 \oplus M\delta_j$.

To see why this solution works, suppose that Alice’s j th circuit is “good” (i.e., garbled correctly and input commitments arranged by Mc_j). As before, define her *effective* OT input to the j th OTs as $c^* = c_j \oplus \delta_j$ (which should be c_1 if Alice did not lie about δ_j). Even if Alice lied about the δ values she surely learns no more than she would have learned by being truthful about the δ values and using effective input c^* in all OTs. Hence, we can imagine that she uses logical input $x_1^* = \hat{x}_1 \oplus Mc^*$ in all of Bob’s garbled circuit.

Alice is required to open garbled inputs indexed by $\hat{x}_1 \oplus M\delta_j = \hat{x}_1 \oplus M(c^* \oplus c_j) = x_1^* \oplus Mc_j$. These are exactly the garbled inputs corresponding to logical input x_1^* , since the commitments were arranged according to Mc_j . We see that Bob evaluates at least one correctly garbled circuit with Alice using input x_1^* , which is all that is required for weak input consistency.

7 Optimizing PSI Reconciliation

7.1 Weaker security.

Our main insight is that our PSI reconciliation step does not require a fully (UC) secure PSI protocol. Instead, a weaker security property suffices. Recall that the final steps of the [16] protocol proceed as follows:

- Alice & Bob commit to their PSI inputs.
- The garbled-output translations are opened and checked.
- The parties either abort or release the PSI output.

For simplicity, assume for now that only one party receives the final PSI output. We will address two-sided output later.

Suppose Alice is corrupt and Bob is honest. Following from the discussion of security in Section 4, Bob will use as PSI input a collection of valid reconciliation values. At the time Alice provides her PSI inputs, the *authenticity* property of the garling scheme is in effect. This means that Alice can predict a valid reconciliation value only for the “correct” output y^* . All other valid reconciliation values that might be part of Bob’s PSI input are unpredictable.

Below we formalize a weak notion of security for input distributions of this form:

Definition 3. Let Π be a two-phase protocol for set intersection ($\mathcal{F}_{psi}^{n,\ell}$, Figure 12). We say that Π is **weakly malicious-secure** if it achieves UC-security with respect to environments that behave as follows:

- (1) The adversary sends a value $a^* \in \{0, 1\}^\ell$ to the environment along with the description of a distribution \mathcal{D} whose support is cardinality- $(n - 1)$ subsets of $\{0, 1\}^\ell$. We further require that \mathcal{D} is **unpredictable** in the sense that the procedure “ $A \leftarrow \mathcal{D}$; output a uniformly chosen element of A ” yields the uniform distribution over $\{0, 1\}^\ell$ (the joint distribution of all elements of A need not be uniform).
- (2) The environment (privately) samples $A \leftarrow \mathcal{D}$ and gives input $A \cup \{a^*\}$ to the honest party for the first phase of PSI.
- (3) After the first phase finishes (i.e., both parties’ inputs are committed), the environment gives the coins used to sample A to the adversary.
- (4) The environment then instructs the honest party to perform the second phase of PSI to obtain output.

In this definition, the adversary knows only one value in the honest party’s set, while all other values are essentially uniform. We claim that when ℓ is large, the simulator for this class of environments *does not need to fully extract the adversary’s PSI input!* Rather, the following are enough to ensure weakly-malicious security:

- The adversary is indeed committed to *some* (unknown to the simulator) effective input during the commit phase.
- The simulator can *test* whether the adversary’s effective PSI input contains the special value a^* .

With overwhelming probability, no effective input element other than a^* can contribute to the PSI output. Any other values in the adversary’s effective input can simply be ignored; they do not need to be extracted.

For technical reasons and convenience in the proof, we have the environment give the adversary the coins used to sample A , but only after the PSI input phase.

7.2 PSZ protocol paradigm.

We now describe an inexpensive protocol paradigm for PSI, due to Pinkas et al. [30]. Their protocol is proven

secure only against passive adversaries. We later discuss how to achieve weak malicious security.

The basic building block is a protocol for **private equality test (PEQT)** based on OT. A benefit of using OT-based techniques is that the bulk of the effort in generating OTs can be done in the offline phase, again leading to a lightweight online phase for the resulting PSI protocol.

Suppose a sender has input s and receiver has input r , with $r, s \in \{0, 1\}^n$, where the receiver should learn whether $r = s$ (and nothing more). The PEQT protocol requires n string OTs; in the i th one, the receiver uses choice bit $r[i]$ and the sender chooses random string inputs (m_0^i, m_1^i) . The sender finally sends $S = \bigoplus_i m_{s[i]}^i$, and the receiver checks whether $S = \bigoplus_i m_{r[i]}^i$, which is the XOR of his OT outputs.

The PEQT can be extended to a **private set membership test (PSMT)**, in which the sender has a set $\{s^1, \dots, s^t\}$ of strings, and receiver learns whether $r \in \{s^1, \dots, s^t\}$. We simply have the sender randomly permute the s^j values, compute for each one $S^j = \bigoplus_i F(m_{s^j[i]}, j)$ and send $\{S^1, \dots, S^t\}$, where F is a PRF.⁵ The receiver can check whether $\bigoplus_i F(m_{r[i]}, j)$ matches S^j for any j . Finally, we can achieve a PSI where the receiver has strings $\{r^1, \dots, r^t\}$ by running independent PSMTs of the form $r^j \in \{s^1, \dots, s^t\}$ for each r^j (in random order).

The overhead of this approach is $O(t^2)$, and [30] describe ways to combine hashing with this basic PSI protocol to obtain asymptotically superior PSI protocols for large sets. However, we are dealing with very small values of t (typically at most 5), so the concrete cost of this simple protocol is very low.

To make the PSI protocol two-phase, we run the OTs and *commit* to the S values in the input-committing phase. Then the output phase consists simply of the sender opening the commitments to S .

7.3 Achieving weakly-malicious security and double-sided output.

We use the [30] protocol but instantiate it with malicious-secure OTs. This leads to the standard notion of security against an active receiver since the simulator can extract the receiver's input from its choice bits to the OTs.

However, the protocol does not achieve full security against a malicious sender. In the simple PEQT building block, the simulator cannot extract a malicious sender's input. Doing so would require inspecting $S, \{m_b^i\}$ and determining a value s such that $S = \bigoplus_i m_{s[i]}^i$. Such an s may not exist, and even if it did, the problem seems

⁵Simply XORing the m_b^i values would reveal some linear dependencies; applying a PRF renders all of the S^j values independently random except the ones for which $r = s^j$.

closely related to a subset-sum problem.

However, if the simulator knows a *candidate* s^* , it can certainly check whether the corrupt sender has sent the corresponding S value. This is essentially the only property required for weakly malicious security.

We note that a corrupt sender could use inconsistent sets $\{s^1, \dots, s^t\}$ in the parallel PSMT instances. However, the simulator can still extract whether the candidate s^* was used in each of them. If the sender used s^* in t' of the t subprotocols, then the simulator can send s^* to the ideal PSI functionality with probability t'/t , which is a sound simulation for weakly-malicious security.

Regarding double-sided output, it suffices to simply run two instances of the one-sided-output PSI protocol, one in each direction, in parallel. Again, this way of composing PSI protocols is not sound *in general*, but it is sound for the special case of weakly-malicious security.

7.4 Trading computation for lower round complexity.

Even when random OTs are pre-processed offline, the PSI protocol as currently described requires two rounds to commit to the outputs, and one round to release the output. The two input-committing rounds are (apparently) inherently sequential, stemming from the sequential nature of OT derandomization.

In terms of round complexity, these two PSI rounds are a bottleneck within the overall dual-execution protocol. We now describe a variant of the PSI protocol in which the two input-committing messages are *asynchronous* and can be sent simultaneously. The modified protocol involves (a nontrivial amount of) additional computation but reduces the number of rounds in the overall 2PC online phase by one. This tradeoff does not *always* reduce the overall latency of the 2PC online phase — only sometimes, depending on the number of garbled circuits being evaluated and the network latency. The specific break-even points are discussed in Section 9.

In our PEQT protocol above, the two parties have pre-processed random OTs, with choice bits c and random strings m_0^i, m_1^i . To commit to his PSI input, the receiver's first message is $d = c \oplus r$, to which the sender responds with $S = \bigoplus_i m_{d[i] \oplus s[i]}^i$.

Consider randomizing the terms of this summation as $S = \bigoplus_i [m_{d[i] \oplus s[i]}^i \oplus z_i]$ where z_i are random subject to $\bigoplus_i z_i = 0$. Importantly, (1) each term in this sum depends only on a single bit of d ; (2) revealing *all* terms in the sum reveals no more than S itself. We let the sender commit to all the potential terms of this sum and reveal them individually in response to d . In more detail, the sender commits to the following values (in this order):

$$(*) \quad [m_{s[1]}^1 \oplus z_1] \quad [m_{s[2]}^2 \oplus z_2] \quad \cdots \quad [m_{s[n]}^n \oplus z_n] \\ [m_{s[1] \oplus 1}^1 \oplus z_1] \quad [m_{s[2] \oplus 1}^2 \oplus z_2] \quad \cdots \quad [m_{s[n] \oplus 1}^n \oplus z_n]$$

Importantly, these commitments can be made before d is known. In response to the message d from the receiver, the sender is expected to release the output by opening the commitments indexed by the bits of d . The sender will open the commitments $\{m_{d[i] \oplus s[i]}^i \oplus z_i\}$; the receiver will compute their XOR S and proceed as before.

The simulator for a corrupt sender simulates a random message d and then checks whether the sender has used a candidate input s^* by extracting the commitments indexed by d to see whether their XOR is $\bigoplus_i m_{d[i] \oplus s^*[i]}^i$.⁶

We can further move the commitments to the offline phase, since there are two commitments per bit of s per PEQT. Observe that the commitments in (\star) are arranged according to the bits of s , which are not known until the online phase. Instead, in the offline phase the sender can commit to these values arranged according to a random string π . In the online phase, the sender commits to its input s by sending $s \oplus \pi$. Then in response to receiver message d , the sender must open the commitments indexed by the bits of $d \oplus (s \oplus \pi)$.

When extending the asynchronous PEQT to a PSMT protocol, the sender commits to an array of $F(m_b^j, j) \oplus z_i^j$ values for each j .

7.5 Final Protocols

For completeness, we provide formal descriptions of the final PSI protocols (synchronous 3-round and asynchronous 2-round) in Figures 13 & 14.

We defer the proof of their security to the full version.

Theorem 4. *The protocols $\Pi_{\text{sync-psi}}$ and $\Pi_{\text{async-psi}}$ described in Figures 13 & 14 are weakly-malicious secure (in the sense of Definition 3) when $\ell \geq \kappa_s$, the statistical security parameter.*

8 Protocol Details & Implementation

The full details of our protocol are given in Figure 15 and the c++ implementation may be found at <https://github.com/osu-crypto/batchDualEx>. The protocol uses three security parameters:

κ_b is chosen so that the protocol will leak a bit to the adversary with probability at most $2^{-\kappa_b}$. This parameter controls the number of garbled circuits used per execution.

κ_s is the statistical security parameter, used to determine the length of the reconciliation strings used as PSI input (the PSI protocol scales with the length of the PSI input values). The adversary can guess an unknown reconciliation value with probability at most $2^{-\kappa_s}$.

⁶Note: although we intend for the two parties' messages to be sent simultaneously, we must be able to simulate in the case that a corrupt sender waits for incoming message d before sending its commitments.

κ_c is the computational security parameter, that controls the key sizes for OTs, commitments and garbled circuits.

In our evaluations we consider $\kappa_c = 128$, $\kappa_s \in \{40, 80\}$ and $\kappa_b \in \{20, 40, 80\}$. In the full version we prove the security of our protocol:

Theorem 5. *Our protocol (Figure 15) securely realizes the $\mathcal{F}_{\text{multi-sfe}}$ functionality, in the presence of malicious adversaries.*

8.1 Implementation & Architecture

In the offline phase, the work is divided between p parallel sets of 4 threads. Within each set, two threads generate OTs and two threads garble and receive circuits and related commitments. Parallelizing OT generation and circuit generation is key to our offline performance; we find that these two activities take roughly the same time.

We generate OTs using an optimized implementation of the Keller et al. [15] protocol for OT extension. Starting from 128 base OTs (computed using the protocol of [28]), we first run an OT extension to obtain $128 \cdot p$ OT instances. We then distribute these instances to the p different thread-sets, and each thread-set uses its 128 OT instances as base OTs to perform its own *independent* OT extension.

We further modified the OT extension protocol to process and finalize OT instances in blocks of 128 instances. This has two advantages: First, OT messages can be used by other threads in the offline phase as they are generated. Second, OT extension involves CPU-bound matrix transposition computations along with I/O-bound communication, and this approach interlaces these operations.

The offline phase concludes by checking the circuits in the cut-and-choose, bucketing the circuits, and exchanging garbled inputs for the random k -probe-encoded inputs.

The online phase similarly uses threading to exploit the inherently parallel nature of the protocol. Upon receiving input, a primary thread sends the other party their input correction value as the first protocol message. This value is in turn given to B sub-threads (where B is the bucket size) that transmit the appropriate wire labels. Upon receiving the labels, the B threads (in parallel) each evaluate a circuit. Each of the B threads then executes (in parallel) one of the set-membership PSI sub-protocols. After the other party has committed to their PSI inputs, the translation tables of each circuit is opened and checked in parallel. The threads then obtain the intersection and the corresponding output value.

8.2 Low-level Optimizations

We instantiate the garbled circuits using the state-of-the-art *half-gates* construction of [35]. The implementation

utilizes the hardware accelerated AES-ni instruction set and uses fixed-key AES as the gate-level cipher, as suggested by [3]. Since circuit garbling and evaluation is the major computation bottleneck, we have taken great care to streamline and optimize the execution pipeline.

The protocol requires the bucket’s common output labels to be random. Instead, we can optimize the online phase choose these labels as the output of a hash at a random seed value. The seed can then be sent instead of sending all of the common output labels. From the seed the other party regenerates the output labels and proceed to validate the output commitments.

9 Performance Evaluation

We evaluated the prototype on Amazon AWS instances c4.8xlarge (64GB RAM and 36 virtual 2.9 GHz CPUs). We executed our prototype in two network settings: a LAN configuration with both parties in the same AWS geographic region and 0.2 ms round-trip latency; and a WAN configuration with parties in different regions and 75 ms round-trip latency.

We demonstrate the scalability of our implementation by evaluating a range of circuits:

- The AES circuit takes a 128-bit key from one party and a 128-bit block from another party and outputs a 128-bit block to both. The circuit consists of 6800 AND gates and 26,816 XOR gates.
- The SHA256 circuit takes 512 bits from both parties, XORs them together and returns the 256-bit hash digest of the XOR’ed inputs. The circuit consists of 90,825 AND gates and 145,287 XOR gates.
- The AES-CBC-MAC circuit takes a 16-block (2048-bit) input from one party and a 128-bit key from the other party and returns the 128-bit result of 16-round AES-CBC-MAC. The circuit consists of 156,800 AND gates and 430,976 XOR gates.⁷

In all of our tests, we use system parameters derived from Lemma 1. N denotes the number of executions, and B denotes the bucket size (number of garbled circuits per execution) and we use $\sim B$ online threads.

9.1 PSI protocol comparison

In Section 7 we describe two PSI protocols that can be used in our 2PC protocol — a *synchronous* protocol that uses three rounds total, and an *asynchronous* protocol that uses two rounds total (at higher communication cost). We now discuss the tradeoffs between these two PSI protocols. A summary is given in Figure 5. For small parameters in the LAN setting, the 2-round asynchronous protocol is faster overall, but for larger parameters the 3-round synchronous protocol is faster. This is

⁷The circuit is not optimized; each call to AES recomputes the entire key schedule.

κ_s	B	PSI		Async		Sync	
		Time	Size	Time	Size	Time	Size
40	2	0.31	2,580	0.35	138		
	3	0.34	5,790	0.39	303		
	4	0.42	10,280	0.46	532		
	6	0.65	32,100	0.55	1,182		
	5	0.55	23,100	0.51	850		
	7	0.83	62,860	0.66	1,638		
80	9	1.39	103,860	0.83	2,682		

Figure 5: The running time (ms) and online communication size (bytes) of the two PSI protocols when executed with κ_s -bit strings and input sets of size B .

$\kappa_s = \kappa_b = 40$		[23]		This		
	Circuit	N	Offline	Online	Offline	Online
LAN	AES	32	197	12	45	1.7
		128	114	10	16	1.5
		1024	74	7	5.1	1.3
	SHA256	32	459	50	136	10.0
		128	275	40	78	8.8
		1024	206	33	48	8.4
WAN	AES	32	1,126	163	282	190
		128	919	164	71	191
		1,024	760	160	34	189
	SHA256	32	3,638	290	777	194
		128	3,426	256	399	192
		1,024	2,992	207	443	191

Figure 6: Amortized running times per execution (reported in ms) for [23] and our prototype. We used bucket size $B = 6, 5, 4$ for $N = 32, 128, 1024$.

due to the extra data sent by the 2-round protocol. Specifically, the asynchronous protocol sends $O(B^2 \kappa_s \kappa_c)$ bytes while the synchronous one sends $O(B \kappa_s + B^2 \kappa_c)$. In the remaining comparisons, we always use the PSI protocol with lowest latency, according to Figure 5.

9.2 Comparison to the LR protocol

We compare our prototype to that of [23] with 40-bit security. That is, we use $\kappa_b = \kappa_s = 40$; both protocols have identical security and use the same bucket size. We use identical AWS instances and a similar number of threads to those reported in [23].

Figure 6 shows the results of the comparison in the LAN setting. It can be seen that our online times are 5 to 7 times faster and our offline times are 4 to 15 times faster. Indeed, for $N = 1024$ our total (online plus offline) time is less than the online time of [23].

In the WAN setting with small circuits such as AES where the input size is minimal we see [23] achieve faster online times. Their protocol has one fewer round than ours protocol, which contributes 38ms to the difference in performance. However, for the larger SHA256 circuit our implementation outperforms that of [23] by 16 to 100ms per execution and we achieve a much more ef-

		$\kappa_b = \kappa_s = 80$			$\kappa_b = \kappa_s = 40$			$\kappa_b = 20; \kappa_s = 40$		
Circuit	N	Storage	Offline	Online	Storage	Offline	Online	Storage	Offline	Online
AES	32	0.21	69	2.3	0.12	45	1.7	0.06	40	1.1
	128	0.88	25	2.1	0.32	16	1.4	0.38	16	1.1
	1,024	6.8	16	1.8	1.6	5.1	1.3	0.76	2.4	1.0
SHA-256	32	6.8	234	15.7	1.3	136	10.0	0.68	65	7.6
	128	8.7	190	12.3	3.5	78	8.8	4.4	95	6.4
	1,024	62.1	131	11.4	15.6	48	8.4	8.8	24	6.3
CBC-MAC	32	3.8	621	22.7	2.4	655	14.9	1.2	247	11.1
	128	15.4	450	18.1	6.2	191	13.4	7.9	246	10.6
	1,024	109.5	378	15.8	31.0	95	12.3	15.6	71	10.6

Figure 7: Amortized running times per execution (reported in ms) and total offline storage (reported in GB) for our prototype in the LAN configuration. The peak offline storage occurs before the cut and choose, consisting of the circuits, commitments, and OT messages. For $\kappa_b = 80$ we use parameters $(N, B) \in \{(32, 12), (128, 9), (1024, 7)\}$. For $\kappa_b = 40$ we use parameters $(N, B) \in \{(32, 6), (128, 5), (1024, 5)\}$. For $\kappa_b = 20$ we use parameters $(N, B) \in \{(32, 3), (128, 2), (1024, 2)\}$.

ficient offline phase ranging from 4 to 22 times faster for both circuits.

As discussed in [Section 2.3](#), our protocol has asymptotically lower online communication cost, especially for computations with larger inputs. Since both protocols are more-or-less I/O bound in these experiments, the difference in communication cost is significant. Concretely, when evaluating AES with $N = 1024$ and $B = 4$ our protocol sends 16,384 bytes of wire labels and just 564 bytes of PSI data. The online phase of [\[23\]](#) reports to use 170,000 bytes with the same parameters. Even using our asynchronous PSI sub-protocol, the total PSI cost is only 10,280 bytes.

9.3 Effect of security parameters

We show in [Figure 7](#) how our prototype scales for different settings of security parameters in the LAN setting. In particular, the security properties of our protocol allow us to consider smaller settings of parameters than are advised with traditional cut-and-choose protocols such as [\[23\]](#). As a representative example, we consider $\kappa_b = 20$ and $\kappa_s = 40$ which means that our protocol will leak a single bit only with probability $1/2^{20}$ but guarantee all other security properties with probability $1 - 1/2^{40}$.

Our protocol scales very well both in terms of security parameter and circuit size. Each doubling of κ_s only incurs an approximate 25% to 50% increase in running time. This is contrasted by [\[23\]](#) reporting a 200% to 300% increase in running time for larger security parameters. Our improvement is largely due to reducing the number of cryptographic steps and no cheat-recovery circuit which consume significant online bandwidth.

We see a more significant trend in the total storage requirement of the offline phase. For example, when performing $N = 1024$ AES evaluations for security parameter $\kappa_b = 20$ the protocol utilizes a maximum of 0.76 GB of storage while $\kappa_b = 40$ requires 1.6 GB of storage. This further validates $\kappa_b = 20$ as a storage and bandwidth saving mechanism. [\[23\]](#) reports that 3.8 GB of offline com-

		LAN		WAN	
κ_s	B	Time	Bandwidth	Time	Bandwidth
40	2	0.26	327	0.63	144
	3	0.41	353	0.72	206
	4	0.56	381	1.01	213
	6	0.82	465	1.32	293
80	5	0.75	568	1.39	300
	7	1.01	725	2.02	366
	9	2.42	465	3.41	331

Figure 8: Maximum amortized throughput (ms/execution) and resulting bandwidth (Kbps) when performing many parallel evaluations of AES with the given bucket size B and statistical security κ_s .

munication for $N = 1024$ and 40-bit security.

9.4 Throughput & Bandwidth

In addition to considering the setting when executions are performed sequentially, we tested our prototype when performing many executions in parallel to maximize *throughput*. [Figure 8](#) shows the maximum average throughput for AES evaluations that we were able to achieve, under different security parameters and bucket sizes. The time reported is the average number of milliseconds per evaluation.

In the LAN setting, 8 evaluations were performed in parallel and achieved an amortized time of 0.26ms per evaluation for bucket size $B = 2$. A bucket size of 2 can be obtained by performing a modest number (say $N = 256$) of executions with $\kappa_b = 20$, or a very large number of executions with $\kappa_b = 40$. We further tested our prototype in the WAN setting where we obtain a slightly decreased throughput of 0.72ms per AES evaluation with 40-bit security.

References

- [1] AFSHAR, A., MOHASSEL, P., PINKAS, B., AND RIVA, B. Non-interactive secure computation based on cut-and-choose. In *EUROCRYPT 2014* (May 2014), P. Q. Nguyen

- and E. Oswald, Eds., vol. 8441 of *LNCS*, Springer, Heidelberg, pp. 387–404.
- [2] BEAVER, D. Precomputing oblivious transfer. In *CRYPTO'95* (Aug. 1995), D. Coppersmith, Ed., vol. 963 of *LNCS*, Springer, Heidelberg, pp. 97–109.
 - [3] BELLARE, M., HOANG, V. T., KEELVEEDHI, S., AND ROGAWAY, P. Efficient garbling from a fixed-key block-cipher. In *2013 IEEE Symposium on Security and Privacy* (May 2013), IEEE Computer Society Press, pp. 478–492.
 - [4] BELLARE, M., HOANG, V. T., AND ROGAWAY, P. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT 2012* (Dec. 2012), X. Wang and K. Sako, Eds., vol. 7658 of *LNCS*, Springer, Heidelberg, pp. 134–153.
 - [5] BELLARE, M., HOANG, V. T., AND ROGAWAY, P. Foundations of garbled circuits. In *ACM CCS 12* (Oct. 2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., ACM Press, pp. 784–796.
 - [6] CANETTI, R. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS* (Oct. 2001), IEEE Computer Society Press, pp. 136–145.
 - [7] CHOU, T., AND ORLANDI, C. The simplest protocol for oblivious transfer. In *Progress in Cryptology - LATINCRYPT 2015* (2015), K. E. Lauter and F. Rodríguez-Henríquez, Eds., vol. 9230 of *Lecture Notes in Computer Science*, Springer, pp. 40–58.
 - [8] DAMGAARD, I., LAURITSEN, R., AND TOFT, T. An empirical study and some improvements of the Mini-Mac protocol for secure computation. *Cryptology ePrint Archive*, Report 2014/289, 2014. <http://eprint.iacr.org/2014/289>.
 - [9] DAMGÅRD, I., PASTRO, V., SMART, N. P., AND ZAKARIAS, S. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012* (Aug. 2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *LNCS*, Springer, Heidelberg, pp. 643–662.
 - [10] DAMGÅRD, I., AND ZAKARIAS, R. W. Fast oblivious AES: a dedicated application of the MiniMac protocol. *Cryptology ePrint Archive*, Report 2015/989, 2015. ia.cr/2015/989.
 - [11] DAMGÅRD, I., AND ZAKARIAS, S. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC 2013* (Mar. 2013), A. Sahai, Ed., vol. 7785 of *LNCS*, Springer, Heidelberg, pp. 621–641.
 - [12] FREDERIKSEN, T. K., JAKOBSEN, T. P., AND NIELSEN, J. B. Faster maliciously secure two-party computation using the GPU. In *SCN 14* (Sept. 2014), M. Abdalla and R. D. Prisco, Eds., vol. 8642 of *LNCS*, Springer, Heidelberg, pp. 358–379.
 - [13] HUANG, Y., KATZ, J., AND EVANS, D. Efficient secure two-party computation using symmetric cut-and-choose. In *CRYPTO 2013, Part II* (Aug. 2013), R. Canetti and J. A. Garay, Eds., vol. 8043 of *LNCS*, Springer, Heidelberg, pp. 18–35.
 - [14] HUANG, Y., KATZ, J., KOLESNIKOV, V., KUMARESAN, R., AND MALOZEMOFF, A. J. Amortizing garbled circuits. In *CRYPTO 2014, Part II* (Aug. 2014), J. A. Garay and R. Gennaro, Eds., vol. 8617 of *LNCS*, Springer, Heidelberg, pp. 458–475.
 - [15] KELLER, M., ORSINI, E., AND SCHOLL, P. Actively secure OT extension with optimal overhead. In *CRYPTO 2015, Part I* (Aug. 2015), R. Gennaro and M. J. B. Robshaw, Eds., vol. 9215 of *LNCS*, Springer, Heidelberg, pp. 724–741.
 - [16] KOLESNIKOV, V., MOHASSEL, P., RIVA, B., AND ROSULEK, M. Richer efficiency/security trade-offs in 2PC. In *TCC 2015, Part I* (Mar. 2015), Y. Dodis and J. B. Nielsen, Eds., vol. 9014 of *LNCS*, Springer, Heidelberg, pp. 229–259.
 - [17] KOLESNIKOV, V., AND SCHNEIDER, T. Improved garbled circuit: Free XOR gates and applications. In *ICALP 2008, Part II* (July 2008), L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórssón, A. Ingólfssdóttir, and I. Walukiewicz, Eds., vol. 5126 of *LNCS*, Springer, Heidelberg, pp. 486–498.
 - [18] KREUTER, B., SHELAT, A., AND SHEN, C. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21th USENIX Security Symposium* (2012), T. Kohno, Ed., USENIX Association, pp. 285–300.
 - [19] LINDELL, Y. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO 2013, Part II* (Aug. 2013), R. Canetti and J. A. Garay, Eds., vol. 8043 of *LNCS*, Springer, Heidelberg, pp. 1–17.
 - [20] LINDELL, Y., AND PINKAS, B. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT 2007* (May 2007), M. Naor, Ed., vol. 4515 of *LNCS*, Springer, Heidelberg, pp. 52–78.
 - [21] LINDELL, Y., AND PINKAS, B. Secure two-party computation via cut-and-choose oblivious transfer. In *TCC 2011* (Mar. 2011), Y. Ishai, Ed., vol. 6597 of *LNCS*, Springer, Heidelberg, pp. 329–346.
 - [22] LINDELL, Y., AND RIVA, B. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In *CRYPTO 2014, Part II* (Aug. 2014), J. A. Garay and R. Gennaro, Eds., vol. 8617 of *LNCS*, Springer, Heidelberg, pp. 476–494.
 - [23] LINDELL, Y., AND RIVA, B. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), I. Ray, N. Li, and C. Kruegel, Eds., ACM, pp. 579–590.
 - [24] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay - secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium* (2004), M. Blaze, Ed., USENIX, pp. 287–302.
 - [25] MOHASSEL, P., AND FRANKLIN, M. Efficiency trade-offs for malicious two-party computation. In *PKC 2006* (Apr. 2006), M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958 of *LNCS*, Springer, Heidelberg, pp. 458–473.
 - [26] MOHASSEL, P., AND RIVA, B. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In *CRYPTO 2013, Part II* (Aug. 2013), R. Canetti and J. A. Garay, Eds., vol. 8043 of *LNCS*, Springer, Heidelberg, pp. 36–53.
 - [27] NIELSEN, J. B., NORDHOLT, P. S., ORLANDI, C., AND BURRA, S. S. A new approach to practical active-secure

- two-party computation. In *CRYPTO 2012* (Aug. 2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *LNCS*, Springer, Heidelberg, pp. 681–700.
- [28] PEIKERT, C., VAIKUNTANATHAN, V., AND WATERS, B. A framework for efficient and composable oblivious transfer. In *CRYPTO 2008* (Aug. 2008), D. Wagner, Ed., vol. 5157 of *LNCS*, Springer, Heidelberg, pp. 554–571.
 - [29] PINKAS, B., SCHNEIDER, T., SMART, N. P., AND WILLIAMS, S. C. Secure two-party computation is practical. In *ASIACRYPT 2009* (Dec. 2009), M. Matsui, Ed., vol. 5912 of *LNCS*, Springer, Heidelberg, pp. 250–267.
 - [30] PINKAS, B., SCHNEIDER, T., AND ZOHNER, M. Faster private set intersection based on OT extension. In *Proceedings of the 23rd USENIX Security Symposium* (2014), K. Fu and J. Jung, Eds., USENIX Association, pp. 797–812.
 - [31] SHELAT, A., AND SHEN, C.-H. Two-output secure computation with malicious adversaries. In *EUROCRYPT 2011* (May 2011), K. G. Paterson, Ed., vol. 6632 of *LNCS*, Springer, Heidelberg, pp. 386–405.
 - [32] SHELAT, A., AND SHEN, C.-H. Fast two-party secure computation with minimal assumptions. In *ACM CCS 13* (Nov. 2013), A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds., ACM Press, pp. 523–534.
 - [33] SMART, N. Personal communication, November 2015.
 - [34] YAO, A. C.-C. Protocols for secure computations (extended abstract). In *23rd FOCS* (Nov. 1982), IEEE Computer Society Press, pp. 160–164.
 - [35] ZAHUR, S., ROSULEK, M., AND EVANS, D. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT 2015, Part II* (Apr. 2015), E. Oswald and M. Fischlin, Eds., vol. 9057 of *LNCS*, Springer, Heidelberg, pp. 220–250.

A Adaptively Secure Garbling Schemes

A **garbling scheme** is a tuple of algorithms $(\text{Gb}, \text{En}, \text{Ev}, \text{De})$ with the following syntax and semantics. All algorithms accept a security parameter as explicit input, which we leave implicit.

- $\text{Gb}(f, d) \rightarrow (F, e)$; Here f is a boolean circuit with m inputs and n outputs; d is an $n \times 2$ array of (output) **wire labels**; F is a **garbled circuit**; and e is an $m \times 2$ array of input wire labels.

By wire labels, we simply mean strings (i.e., elements of $\{0, 1\}^{\kappa_c}$). We deviate from [5] in requiring the output wire labels d to be chosen by the caller of Gb , rather than chosen by Gb itself. In the notation of [5], we assume that the scheme is **projective** in both its input and output encodings, meaning that e and d consist of two possible wire labels for each wire.

- $\text{En}(e, x) \rightarrow X$ takes an $m \times 2$ array of wire labels e and a plaintext input $x \in \{0, 1\}^m$ and outputs a **garbled encoding** X of x . By assuming that the scheme is projective, we assume that $X = (X_1, \dots, X_m)$ where $X_i = e[i, x_i]$.

- $\text{Ev}(F, X) \rightarrow Y$; takes a garbled circuit F and garbled encoding X of an input, and returns a garbled encoding of the output Y .

- $\widetilde{\text{De}}(Y) \rightarrow y$. We assume a way to decode a garbled output to a plaintext value. It is a deviation from [5] to allow this to be done without the decoding information d . Rather, we may assume that the garbled outputs contain the plaintext value, say, as the last bit of each wire label.

Our correctness condition is that for the variables defined above, we have $\text{Ev}(F, \text{En}(e, x)) = \text{En}(d, f(x))$ and $\widetilde{\text{De}}(\text{Ev}(F, \text{En}(e, x))) = f(x)$ for all inputs x to the circuit f . In other words, evaluating the garbled circuit should result in the garbled output that encodes $f(x)$ under the encoding d .

In our construction, an adversary sees the garbled circuit F first, then it receives some of the garbled inputs (corresponding to the k -probe matrix encoded inputs). Finally in the online phase it is allowed to choose the rest of its input to the circuit and receive the rest of the garbled inputs. Hence, our security game considers an adversary that can obtain the information in this order.

We overload the syntax of the encoding algorithm En . Since En is projective, we write $\text{En}(e, i, b)$ to denote the component $e_{i,b}$ — that is, the garbled input for the i th wire corresponding to truth value b . Recall that we also garble a circuit with output wire labels d specified (rather than chosen by the Gb algorithm). Our security definition lets the adversary choose d .

Definition 6. For a garbling scheme $(\text{Gb}, \text{En}, \text{Ev}, \text{De})$, an interactive oracle program Adv , and algorithms $S = (S_0, S_1, S_2)$, we define the following two games/interactions:

$\mathcal{G}_{\text{ideal}}^{\text{Adv}, S}$:
get f, d from Adv^{S_0}
$F \leftarrow S_1(f)$
give F to Adv^{S_0}
for $i = 1$ to $m - 1$:
get x_i from Adv^{S_0}
$X_i \leftarrow S_2(i)$
give X_i to Adv^{S_0}
get x_m from Adv^{S_0}
$y = f(x_1 \dots x_m)$
$Y \leftarrow \text{En}(d, y)$
$X_m \leftarrow S_2(m, y, Y)$
give X_m to Adv^{S_0}
Adv^{S_0} outputs a bit

In $\mathcal{G}_{\text{ideal}}$, H is a random oracle. In $\mathcal{G}_{\text{ideal}}$, the tuple $S = (S_0, S_1, S_2)$ all share state. All algorithms receive the security parameter as implicit input.

Then the garbling scheme is **adaptively secure** if there exists a simulator S such that for all polynomial-time adversaries Adv , we have that

$$|\Pr[\mathcal{G}_{\text{real}}^{\text{Adv}} \text{ outputs } 1] - \Pr[\mathcal{G}_{\text{ideal}}^{\text{Adv}, S} \text{ outputs } 1]|$$

is negligible in the security parameter.

Note that in the $\mathcal{G}_{\text{ideal}}$ game, the simulator receives no information about the input x as it produces the garbled circuit F and all but one of the garbled input components. Finally when producing the last garbled input component, the simulator learns $f(x)$ and its *garbled output encoding* $\text{En}(d, f(x))$. In particular, the simulator receives no information about x , so its outputs carry no information about x beyond $f(x)$. The game also implies an authenticity property for garbled outputs of values other than $f(x)$ — the simulator’s total output contains no information about the rest of the garbled outputs d .

In [Figure 9](#) we describe a generic, random-oracle transformation from a standard (static-secure) garbling scheme to one with this flavor of adaptive security. The construction is quite similar to the transformations in [\[4\]](#), with some small changes. First, since we know in advance which order the adversary will request its garbled inputs, we include the random oracle nonce R in the last garbled input value (rather than secret-sharing across all garbled inputs). Second, since we garble a circuit with particular garbled output values in mind, we provide “translation values” that will map the garbled outputs of the static scheme to the desired ones. These translation values also involve the random oracle, so they can be equivocated by the simulator.

Theorem 7. If $(\text{Gb}, \text{En}, \text{Ev}, \text{De}, \widetilde{\text{De}})$ is a doubly-projective garbling scheme satisfying the (static) prv and aut properties of [\[5\]](#) then the scheme in [Figure 9](#) satisfies adaptive security notion of [Definition 6](#) in the random oracle model.

The proof is very similar to analogous proofs in [\[4\]](#). The main idea is that the simulator can choose the “masked” \widehat{F} and δ translation values upfront. Then it is only with negligible probability that an adversary will call the random oracle on the secret nonce R , so the relevant parts of the oracle are still free to be programmed by the simulator. When the adversary provides the final bit of input, the simulator gets $f(x)$ and can obtain a simulated garbled circuit \widehat{F} and garbled outputs \widehat{d} from the static-secure scheme. Then it can program the random oracle to return the appropriate masks.⁸

$\widehat{\text{Gb}}(f, \widehat{d}):$ $(F, e, d) \leftarrow \text{Gb}(f)$ $R \leftarrow \{0, 1\}^\kappa$ $\text{for each output wire } i:$ $\delta_i^b \leftarrow H(R \parallel \text{out} \parallel i \parallel b \parallel d_i^b) \oplus \widehat{d}_i^b$ $\widehat{F} \leftarrow (F \oplus H(R \parallel \text{gc}), \{\delta_i^b\})$ $\widehat{e} \leftarrow (e_1^0, e_1^1, e_2^0, e_2^1, \dots, e_m^0 \parallel R, e_m^1 \parallel R)$ $\text{return } (\widehat{F}, \widehat{e})$
$\widehat{\text{Ev}}(\widehat{F}, \widehat{X}):$ $\text{parse } \widehat{X}_m \text{ as } X_m \parallel R \text{ and } \widehat{F} \text{ as } (F', \delta)$ $X \leftarrow (\widehat{X}_1, \widehat{X}_2, \dots, X_m)$ $Y \leftarrow \text{Ev}(F' \oplus H(R \parallel \text{gc}), X)$ $y \leftarrow \widetilde{\text{De}}(Y)$ $\text{for each output wire } i:$ $\widehat{Y}_i = \delta_i^{y_i} \oplus H(R \parallel \text{out} \parallel i \parallel y_i \parallel Y_i)$ $\text{return } \widehat{Y}$

Figure 9: Transformation from a static-secure doubly-projective garbling scheme $(\text{Gb}, \text{En}, \text{Ev}, \text{De}, \widetilde{\text{De}})$ to one satisfying [Definition 6](#).

⁸Technically, the proof assumes that the simulator for the static-secure scheme can set the (simulated) garbled input encoding arbitrarily. This is true for common existing schemes; e.g., [\[35\]](#).

Setup stage: On common input $(\text{sid}, \text{SETUP}, f, N, \epsilon)$, where f is a boolean circuit, N is the number of executions. The parties agree on parameters B, \hat{N} derived from [Lemma 1](#). Let $M \in \{0, 1\}^{\mu \times n}$ be a κ_s -probe resistant matrix for each party's input of size n . Let $a \in \{0, 1\}$ denote the role of the current party and $b = a \oplus 1$. Note: the protocol is symmetric where both parties simultaneously play the roles of P_a and P_b .

- **Cut-and-Choose Commit:** P_a chooses at random the cut and choose set $\sigma_a \subset [\hat{N}]$ of size $\hat{N} - NB$. P_a send $(\text{COMMIT}, (\text{sid}, \text{CUT-AND-CHOOSE}, a), \sigma_a)$ to \mathcal{F}_{com} . For $j \in [\hat{N}]$:
 - **OT Init:** P_a sends $(\text{INIT}, (\text{sid}, \text{OT}, a, j))$ to $\mathcal{F}_{\text{rot}}^\mu$ and receives choice bits c_j^a in response.
 - **Send Circuit:** P_a chooses random output wire labels d_j , computes $(F_j^a, e_j) \leftarrow \widehat{\text{Gb}}(f', d_j)$ and sends the F_j^a to P_b where $f'(x_a, r, \tilde{x}_b) = f(x_a, Mr \oplus \tilde{x}_b)$ and r, \tilde{x}_b are P_b 's inputs. Let e_j^a, e_j^b, e_j^r respectively be the labels encoding x_a, \tilde{x}_b, r , for circuit F_j^a and $e_{j,t,h}^*$ index the label of the t^{th} wire with value h in the set e_j^*
 - **Input Commit:** P_a sends the following to \mathcal{F}_{com} :
 - ▶ $(\text{COMMIT}, (\text{sid}, x_a\text{-INPUT}, a, j, t, h), e_{j,t,Mc[t] \oplus h}^a)_{t \in [n], h \in \{0,1\}}$.
 - ▶ $(\text{COMMIT}, (\text{sid}, x_b\text{-INPUT}, a, j, t, h), e_{j,t,h}^b)_{t \in [n], h \in \{0,1\}}$
 - ▶ $(\text{COMMIT}, (\text{sid}, r\text{-INPUT}, a, j, t, h), e_{j,t,h}^r)_{t \in [\mu], h \in \{0,1\}}$
 - **Output Commit:** P_a sends $(\text{COMMIT}, (\text{sid}, \text{OUTPUT}, a, j), d_j)$ to \mathcal{F}_{com} .
- **Cut-and-Choose:** P_b sends $(\text{OPEN}, (\text{sid}, \text{CUT-AND-CHOOSE}, b))$ to \mathcal{F}_{com} and P_a receives σ_b . For $j \in \sigma_b$:
 - **OT Decommit:** P_a sends $(\text{OPEN}, (\text{sid}, \text{OT}, a, j))$ to $\mathcal{F}_{\text{rot}}^\mu$ and P_b receives choice bits c_j^b .
 - **Check Circuit:** P_a sends P_b the d_j and coins used to garble F_j^a . P_b verifies the correctness of F_j^a .
 - **Input Decommit:** Let e_j^a, e_j^b, e_j^r be the verified labels as above.
 - ▶ P_a sends $(\text{OPEN}, (\text{sid}, x_a\text{-INPUT}, a, j, t, h))_{\forall t, h}$ to \mathcal{F}_{com} and P_b receives labels e^a .
 - ▶ P_a sends $(\text{OPEN}, (\text{sid}, x_b\text{-INPUT}, a, j, t, h))_{\forall t, h}$ to \mathcal{F}_{com} and P_b receives labels e^b .
 - ▶ P_a sends $(\text{OPEN}, (\text{sid}, r\text{-INPUT}, a, j, t, h))_{\forall t, h}$ to \mathcal{F}_{com} and P_b receives labels e^r .
 - ▶ If there exists a $e_{j,t,h}^a \neq e_{j,t,Mc[t] \oplus h}^a$, or $e_{j,t,h}^b \neq e_j^b$, or $e_{j,t,h}^r \neq e_j^r$, P_b returns ABORT .
 - **Output:** P_a sends $(\text{OPEN}, (\text{sid}, \text{OUTPUT}, a, j))$ to \mathcal{F}_{com} . P_b receives d' and return ABORT if $d' \neq d_j$.
- **Bucketing:** P_b randomly maps the indices of $[\hat{N}] - \sigma_b$ into sets $\beta_1^a, \dots, \beta_N^a$ s.t. $|\beta_i^a| = B$. For $i \in [N]$:
 - **Bucket Labels:** P_a generates random output labels O_i^a for bucket β_i^a . For $j \in \beta_i^a$, P_a send the output translation $T_j^a := \{O_{i,t,h}^a \oplus d_{j,t,h}\}_{t,h}$ and $H(O_{i,t,h}^a)$ to P_b , where d_j are the output labels of F_j^a .
 - **Offline Inputs:**
 - ▶ P_a sends $(\text{AGGREGATE}, (\text{sid}, \text{OT-AG}, a, i), \{(\text{sid}, \text{OT}, a, j) | j \in \beta_i^a\})$ to $\mathcal{F}_{\text{rot}}^\mu$ and P_b receives the OT aggregation strings δ_j^a for $j \in \beta_i^a$.
 - ▶ P_b sends $(\text{DELIVER}, (\text{sid}, \text{OT-AG}, a, i), \{e_j^r, w_j | j \in \beta_i^a\})$ to $\mathcal{F}_{\text{rot}}^\mu$ where w_j are the decommitment strings to $\{(\text{sid}, r\text{-INPUT}, b, j, t, h)\}_{t,h}$.
 - ▶ For $j \in \beta_i^a$, P_a receives X_j^r and W_j from $\mathcal{F}_{\text{rot}}^\mu$. P_a send $(\text{OPEN}, (\text{sid}, r\text{-INPUT}, b, j, t, c_j^a[t]), W_{j,t})_{\forall t}$ to \mathcal{F}_{com} and receives $X_j'^r$. P_a returns ABORT if $X_j'^r \neq X_j^r$.

Execution stage: On common bucket index i and P_a 's input x_a .

- **Receiver's Inputs:** Let j' be the first index in β_i^a . P_a sends $\tilde{x}_a := x_a \oplus Mc_{j'}^a$ to P_b where $c_{j'}^a$ are the choice bits of $(\text{sid}, \text{OT}, a, j')$. For all $j \in \beta_i^b$:
 - P_b sends $X_j^a := \{e_{j,t,(\tilde{x}_a \oplus M\delta_j^a)[t]}^a\}_t$ and $W_j^a := \{w_{j,t,(\tilde{x}_a \oplus M\delta_j^a)[t]}^a\}_t$ to P_a where e_j^a encodes \tilde{x}_a for F_j^b and w_j are the decommitments string to $\{(\text{sid}, x_a\text{-INPUT}, b, j, t, h)\}_{t,h}$.
 - P_a receives X_j^a, W_j^a and sends $(\text{OPEN}, (\text{sid}, x_a\text{-INPUT}, b, j, t, (x_a \oplus Mc_j^a)[t]), W_{j,t}^a)_{\forall t}$ to \mathcal{F}_{com} and receives $X_j'^a$. P_a returns ABORT if $X_j'^a \neq X_j^a$.
- **Sender's Inputs:** For $j \in \beta_i^b$, P_b sends $(\text{OPEN}, (\text{sid}, x_b\text{-INPUT}, b, j, t, (x_b \oplus Mc_j^b)[t]))_{\forall t}$ to \mathcal{F}_{com} . P_a receives the labels X_j^b . P_a returns ABORT if $\tilde{x}_b \oplus M\delta_j^b \neq (x_b \oplus Mc_j^b)$.
- **Evaluate:** For $j \in \beta_i^b$, let $Y_j := \widehat{\text{Ev}}(F_j^b, (X_j^b, X_j^r, X_j^a))$ with semantic value y_j .
- **PSI Commit:** For $\forall j, t$, if $(H(Y_{j,t}) \neq H(O_{i,t,y_j[t]}^b))$, then $Y_{j,t} \leftarrow \{0, 1\}^{\kappa_c}$. Let $I := \{\bigoplus_t Y_{j,t} \oplus T_{j,t,y_j[t]}^b \oplus O_{i,t,y_j[t]}^a\}_{j \in \beta_i^b}$. P_a pads I to size B with random values and sends $(\text{INPUT}, (\text{sid}, \text{PSI}, i), I)$ to \mathcal{F}_{psi} and receives (INPUT, P_b) .
- **Output Decommit:** For $j \in \beta_i^b$, P_b sends $(\text{OPEN}, (\text{sid}, \text{OUTPUT}, b, j))$ to \mathcal{F}_{com} and P_a receives d'_j .
If there exists j, j' s.t. $d'_{j,t,h} \oplus T_{j,t,h}^b \neq d'_{j',t,h} \oplus T_{j',t,h}^b$, P_a returns ABORT.
- **PSI Decommit:** P_b sends $(\text{OPEN}, (\text{sid}, \text{PSI}, i))$ to \mathcal{F}_{psi} and P_a receives the intersection R . If $|R| \neq 1$, P_a returns CHEATING!, else P_A returns y_j s.t. $I_j \in R$.

Figure 15: Malicious secure online/offline dual-execution 2PC protocol $\Pi_{\text{multi-sfe}}$.

Parameters: A sender P_1 and receiver P_2 .

Setup: On common input S from both parties, for every $s \in S$ choose random $m_0, m_1 \leftarrow \{0, 1\}^{\kappa_c}$ and random $c \leftarrow \{0, 1\}$. Internally store a tuple (s, m_0, m_1, c) .

P_1 output: On input (GET, s) from P_1 , if there is a tuple (s, m_0, m_1, c) for some m_0, m_1, c then give $(OUTPUT, s, m_0, m_1)$ to P_1 .

P_2 output: On input (GET, s) from P_2 , if there is a tuple (s, m_0, m_1, c) for some m_0, m_1, c then give $(OUTPUT, s, c, m_c)$ to P_2 .

Figure 10: Random OT functionality \mathcal{F}_{ot} .

Parameters: A sender P_1 and receiver P_2 .

Commit: On input $(COMMIT, \text{sid}, v)$ from P_1 : If a tuple of the form $(\text{sid}, \cdot, \cdot)$ is stored, then abort. If P_1 is corrupt, then obtain value r from the adversary; otherwise choose $r \leftarrow \{0, 1\}^{\kappa_c}$ and give r to P_1 . Internally store a tuple (sid, r, v) and give $(COMMITTED, \text{sid})$ to P_2 .

Reveal: On input $(OPEN, \text{sid}, r')$ from P_2 : if a tuple (sid, r', v) is stored for some v , then give $(OPENED, \text{sid}, v)$ to P_2 . Otherwise, give $(ERROR, \text{sid})$ to P_2 .

Figure 11: Non-interactive commitment functionality \mathcal{F}_{com} .

Parameters: Two parties: a sender P_1 and receiver P_2 ; $\ell =$ length of items; $n =$ size of parties' sets.

First phase (input commitment): On input $(INPUT, A_i)$ from party P_i ($i \in \{1, 2\}$), with $A_i \subseteq \{0, 1\}^\ell$ and $|A_i| = n$: If this is the first such command from P_i then internally record A_i and send message $(INPUT, P_i)$ to both parties.

Second phase (output): On input $(OUTPUT)$ from P_i , deliver $(OUTPUT, A_1 \cap A_2)$ to the other party.

Figure 12: Two-phase private set intersection (PSI) functionality $\mathcal{F}_{\text{psi}}^{n, \ell}$.

Parameters: Two parties: a sender P_1 and receiver P_2 ; $\ell =$ bit-length of items in the set; $n =$ size of parties' sets; $F =$ a PRF.

Offline phase: Parties perform random OTs, resulting in P_1 holding strings $m_{\{0,1\}}^{i,t} \leftarrow \{0, 1\}^{\kappa_c}$; and P_2 holding c_i and $m_{c_i[t]}^{i,t}$. Here, $c_i \in \{0, 1\}^\ell$ and $i \in [n], t \in [\ell]$.

Input committing phase:

- On input $(INPUT, \{A_{2,1}, \dots, A_{2,n}\})$ to P_2 , P_2 randomly permutes its input and then sends $d_i := A_{2,i} \oplus c_i$ for each $i \in [n]$.
- On input $(INPUT, \{A_{1,1}, \dots, A_{1,n}\})$ for P_1 , P_1 randomly permutes its input and then computes $S_{i,j} = \bigoplus_t F(m_{d_i[t] \oplus A_{1,j}[t]}^{i,t}, j)$ for $i, j \in [n]$.
- P_1 sends $(COMMIT, \text{sid}, (S_{1,1}, \dots, S_{n,n}))$ to \mathcal{F}_{com} .

Output phase: On input $(OUTPUT)$, P_1 sends $(OPEN, \text{sid})$ to \mathcal{F}_{com} and P_2 receives $(OPENED, \text{sid}, (S_{1,1}, \dots, S_{n,n}))$. P_2 then outputs $\{A_{2,i} \mid \exists j : \bigoplus_t F(m_{c_i[t]}^{i,t}, j) = S_{i,j}\}$.

Figure 13: Weakly-malicious-secure, synchronous (3-round), two-phase PSI protocol $\Pi_{\text{sync-psi}}$.

Parameters: Two parties: a sender P_1 and receiver P_2 ; $\ell =$ bit-length of items in the set; $n =$ size of parties' sets; $F =$ a PRF.

Offline phase: Parties perform random OTs, resulting in P_1 holding strings $m_{\{0,1\}}^{i,t} \leftarrow \{0, 1\}^{\kappa_c}$; and P_2 holding c_i and $m_{c_i[t]}^{i,t}$. Here, $c_i \in \{0, 1\}^\ell$ and $i \in [n], t \in [\ell]$.

For $i \in [n]$, P_1 chooses $\pi_i \leftarrow \{0, 1\}^\ell$. Then for $i, j \in [n]$, party P_1 does the following:

- For $t \in \{0, 1\}^\ell$, choose $z_t^{i,j} \leftarrow \{0, 1\}^\ell$ subject to $\bigoplus_t z_t^{i,j} = 0$
- for $t \in [\ell], b \in \{0, 1\}$; P_1 sends $(COMMIT, (\text{sid}, i, j, t, b), F(m_{\pi_j[t] \oplus b}^{i,t}, j) \oplus z_t^{i,j})$ to \mathcal{F}_{com} .

Input committing phase: On input $(INPUT, \{A_{1,1}, \dots, A_{1,n}\})$ for P_1 and $(INPUT, \{A_{2,1}, \dots, A_{2,n}\})$ for P_2 , the parties randomly permute their inputs and asynchronously do:

- P_1 sends $d_{1,j} := A_{1,j} \oplus \pi_j$ for each $j \in [n]$
- P_2 sends $d_{2,i} := A_{2,i} \oplus c_i$ for each $i \in [n]$

Output phase: On input $(OUTPUT)$: for $i, j \in [n]$, $t \in [\ell]$, party P_1 sends $(OPEN, (\text{sid}, i, j, t, d_{1,j}[t] \oplus d_{2,i}[t]))$ to \mathcal{F}_{com} and P_2 expects to receive $(OPENED, (\text{sid}, i, j, t, d_{1,j}[t] \oplus d_{2,i}[t]), \rho_t^{i,j})$.

P_2 outputs $\{A_{2,i} \mid \exists j : \bigoplus_t F(m_{c_i[t]}^{i,t}, j) = \bigoplus_t \rho_t^{i,j}\}$

Figure 14: Weakly-malicious-secure, asynchronous (2-round), two-phase PSI protocol $\Pi_{\text{async-psi}}$.

Egalitarian computing

Alex Biryukov

University of Luxembourg

alex.biryukov@uni.lu

Dmitry Khovratovich

University of Luxembourg

khovratovich@gmail.com

Abstract

In this paper we explore several contexts where an adversary has an upper hand over the defender by using special hardware in an attack. These include password processing, hard-drive protection, cryptocurrency mining, resource sharing, code obfuscation, etc.

We suggest memory-hard computing as a generic paradigm, where every task is amalgamated with a certain procedure requiring intensive access to RAM both in terms of size and (very importantly) bandwidth, so that transferring the computation to GPU, FPGA, and even ASIC brings little or no cost reduction. Cryptographic schemes that run in this framework become *egalitarian* in the sense that both users and attackers are equal in the price-performance ratio conditions.

Based on existing schemes like Argon2 and the recent generalized-birthday proof-of-work, we suggest a generic framework and two new schemes:

- MTP, a memory-hard Proof-of-Work based on the memory-hard function with fast verification and short proofs. It can be also used for memory-hard time-lock puzzles.
- MHE, the concept of memory-hard encryption, which utilizes available RAM to strengthen the encryption for the low-entropy keys (allowing to bring back 6 letter passwords).

Keywords: MTP, MHE, Argon2, memory-hard, asymmetric, proof-of-work, botnets, encryption, time-lock puzzles.

1 Introduction

1.1 Motivation

Historically attackers have had more resources than defenders, which is still mostly true. Whether it is secret key recovery or document forgery, the attackers are

ready to spend tremendous amount of computing power to achieve the goal. In some settings it is possible to make most attacks infeasible by simply setting the key length to 128 bits and higher. In other settings the secret is limited and the best the defender can do is to increase the time needed for the attack, but not to render the attack impossible.

Passwords, typically stored in a hashed form, are a classical example. As people tend to choose passwords of very low entropy, the security designers added unique salts and then increased the number of hash iterations. In response the attackers switched to dedicated hardware for password cracking, so that the price of single password recovery dropped dramatically, sometimes by a few orders of magnitude.

A similar situation occurred in other contexts. The Bitcoin cryptocurrency relies on continuous preimage search for the SHA-256 hash function, which is much cheaper on custom ASICs, consuming up to 30,000 times less energy per solution than most efficient x86 laptops [2]. Eventually, the original concept of an egalitarian cryptocurrency [25] vanished with the emergence of huge and centralized mining pools.

Related problems include password-based key derivation for hard-drive encryption, where the data confidentiality directly depends on the password entropy, and where offline attack is exceptionally easy once the drive is stolen. Similar situation arise in the resource sharing and spam countermeasures. In the latter it is proposed that every user presents a certain proof (often called proof-of-work), which should be too expensive for spammers to generate on a large scale. Yet another setting is that of code obfuscation, in which powerful reverse-engineering/de-compilation tools can be used in order to lift the proprietary code or secrets embedded in the software.

1.2 Egalitarian computing

Our idea is to remedy the disparity between ordinary users and adversaries/cheaters, where latter could use botnets, GPU, FPGA, ASICs to get an advantage and run a cheaper attack. We call it *egalitarian computing* as it should establish the same price for a single computation unit on all platforms, so that the defender's hardware is optimal both for attack and defence. Equipped with egalitarian crypto schemes, defenders may hope to become to be on par with the most powerful attackers.

The key element of our approach is large (in size) and intensive (in bandwidth) use of RAM as a widely available and rather cheap unit for most defenders. In turn, RAM is rather expensive on FPGA and ASIC¹, and slow on GPU, at least compared to memoryless computing tasks. All our schemes use a lot of memory and a lot of bandwidth — almost as much as possible.

We suggest a single framework for this concept and concrete schemes with an unique combination of features.

In the future, adoption of our concept could allow a homogenization of computing resources, a simplified security analysis, and relaxed security requirements. When all attackers use the same hardware as defenders, automated large-scale attacks are no longer possible. Shorter keys, shorter passwords, faster and more transparent schemes may come back to use.

Related work The idea of extensive memory use in the context of spam countermeasures dates back at least to 2003 [5, 13] and was later refined in [15]. Fast memory-intensive hash functions were proposed first by Percival in 2009 [27] and later among the submissions of the Password Hashing Competition. Memory-intensive proofs-of-work have been studied both in theory [16] and practice [6, 32].

Paper structure We describe the goals of our concept and give a high level overview in Section 2. Then we describe existing applications where this approach is implicitly used: password hashing and cryptocurrency proofs of work (Section 3). We present our own progress-free Proof-of-Work MTP with fast verification, which can also serve as a memory-hard time-lock puzzle, in Section 4. The last Section 5 is devoted to the

¹The memory effect on ASICs can be illustrated as follows. A compact 50-nm DRAM implementation [17] takes 500 mm² per GB, which is equivalent to about 15000 10 MHz SHA-256 cores in the best Bitcoin 40-nm ASICs [1] and is comparable to a CPU size. Therefore, an algorithm requiring 1 GB for 1 minute would have the same AT cost as an algorithm requiring 2⁴² hash function calls, whereas the latter can not finish on a PC even in 1 day. In other words, the use of memory can increase the AT cost by a factor of 1000 and more at the same time cost for the desktop user.

novel concept of memory-hard encryption, where we present our scheme MHE aimed to increase the security of password-based disk encryption.

2 Egalitarian computing as framework

2.1 Goal

Our goal is to alter a certain function \mathcal{H} in order to maximize its computational cost on the most efficient architecture – ASICs, while keeping the running time on the native architecture (typically x86) the same. We ignore the design costs due to nontransparent prices, but instead estimate the running costs by measuring the time-area product [8, 31]. On ASICs the memory size M translates into certain area A . The ASIC running time T is determined by the length of the longest computational chain and by the ASIC memory latency.

Suppose that an attacker wants to compute \mathcal{H} using only a fraction αM of memory for some $\alpha < 1$. Using some tradeoff specific to \mathcal{H} , he has to spend $C(\alpha)$ times as much computation and his running time increases by the factor $D(\alpha)$ (here $C(\alpha)$ may exceed $D(\alpha)$ as the attacker can parallelize the computation). In order to fit the increased computation into time, the attacker has to place $\frac{C(\alpha)}{D(\alpha)}$ additional cores on chip. Therefore, the time-area product changes from AT_1 to AT_α as

$$AT_\alpha = A \cdot (\alpha + \frac{\beta C(\alpha)}{D(\alpha)}) T \cdot D(\alpha) = \\ = AT_1 (\alpha D(\alpha) + C(\alpha)\beta), \quad (1)$$

where β is the fraction of the original memory occupied by a single computing core. If the tradeoff requires significant communication between the computing cores, the memory bandwidth limit Bw_{max} may also increase the running time. In practice we will have $D(\alpha) \geq C(\alpha) \cdot Bw/Bw_{max}$, where Bw is the bandwidth for $\alpha = 1$.

Definition 1 We call function \mathcal{F} memory-hard (w.r.t. M) if any algorithm \mathcal{A} that computes \mathcal{H} using αM memory has the computation-space tradeoff $C(\alpha)$ where $C()$ is at least a superlinear function of $1/\alpha$.

It is known [19] that any function whose computation is interpreted as a directed acyclic graph with T vertices of constant in-degree, can be computed using $O(\frac{T}{\log T})$ space, where the constant in $O()$ depends on the degree. However, for concrete hash functions very few tradeoff strategies have been published, for example [9].

2.2 Framework

Our idea is to combine a certain computation \mathcal{H} with a memory-hard function \mathcal{F} . This can be done by modifying \mathcal{H} using input from \mathcal{F} (*amalgamation*) or by transforming its code to an equivalent one (*obfuscation*).

The **amalgamation** is used as follows. The execution of \mathcal{H} is typically a sequence of smaller steps $H_i, i < T$, which take the output V_{i-1} from the previous step and produce the next output V_i . For our purpose we need another primitive, a memory-hard function \mathcal{F} , which fills the memory with some blocks $X[i], i < T$. We suggest combining \mathcal{H} with \mathcal{F} , for example like:

$$\mathcal{H}' = H'_T \circ H'_{T-1} \circ \cdots \circ H'_1,$$

where

$$H'_i(V_{i-1}) = H(V_{i-1} \oplus X[i-1]).$$

Depending on the application, we may also modify $X[i]$ as a function of V_{i-1} so that it is impossible to precompute \mathcal{F} . The idea is that any computation of \mathcal{H}' should use T blocks of memory, and if someone wants to use less, the memory-hardness property would impose computational penalties on him. This approach will also work well for any code that uses nonces or randomness produced by PRNG. PRNG could then be replaced by (or intermixed with the output of) F .

The **obfuscation** principle works as follows. Consider a compiler producing an assembly code for some function \mathcal{H} . We make it to run a memory-hard function \mathcal{F} on a user-supplied input I (password) and produce certain number of memory blocks. For each if-statement of the form

```
if x then A
else B
```

the compiler computes a *memory-hard bit* b_i which is extracted from the block $X[i]$ (the index can also depend on the statement for randomization) and alters the statement as

```
if x ⊕ bi then A
else B
```

for $b_i = 0$ and

```
if x ⊕ bi then B
else A
```

for $b_i = 1$. This guarantees that the program will have to run \mathcal{F} at least once (the bits b_i can be cached if this if-statement is used multiple times, ex. in a loop).

Accessing the memory block from a random memory location for each conditional statement in practice would slow down the program too much, so compiler can perform a tradeoff depending on the length of the program, the number of conditional statements in it and according to a tunable degree of required memory-hardness for a program. Memory-hard bits could be mixed into opaque predicates or other code obfuscation constructs like code-flattening logic.

We note that in order for a program to run correctly, the user needs to supply correct password for \mathcal{F} , even though the source code of the program is public. A smart decompiler, however, when supplied with the password, can obtain clean version of the program by running \mathcal{F} only once.

Our schemes described in the further text use the amalgamation principle only, so we leave the research directions in obfuscation for future work.

3 Egalitarian computing in applications

In this section we outline several applications, where memory-hard functions are actively used in order to achieve egalitarian computing.

3.1 Password hashing with a memory-hard function

The typical setting for the password hashing is as follows. A user selects a password P and submit it to the authentication server with his identifier U . The server hashes P and unique salt S with some function \mathcal{F} , and stores $(U, S, F(P, S))$ in the password file. The common threat is the password file theft, so that an attacker can try the passwords from his dictionary file D and check if any of them yields the stolen hash. The unique S ensures that the hashes are tried one-by-one.

Following massive password cracking attacks that use special hardware [23, 30], the security community initiated the Password Hashing Competition [3] to select the hash function that withstands the most powerful adversaries. The Argon2 hash function [10] has been recently selected as the winner. We stress that the use of memory-hard function for password hashing does not make the dictionary attacks infeasible, but it makes them much more expensive in terms of the single trial cost.

Definition and properties of Argon2 We use Argon2 in our new schemes described in Sections 4 and 5. Here we outline the key elements of the Argon2 design that are used in our scheme. For more details and their rationale we refer the reader to [10].

Argon2 takes P , S , and possibly some additional data U as inputs. It is parametrized by the memory size M , number of iterations t , and the available parallelism l . It fills M blocks of memory $X[1], X[2], \dots, X[M]$ (1 KB each) and then overwrites them $(t - 1)$ times. Each block $X[i]$ is generated using internal compression function F , which takes $X[i-1]$ and $X[\phi(i)]$ as inputs. For $t = 1$ this works as follows, where H is a cryptographic hash

function (Blake2b).

$$\begin{aligned} X[1] &= H(P, S); \\ X[i] &= F(X[i-1], X[\phi(i)]), i > 1; \\ Out &\rightarrow H(X[M]). \end{aligned} \quad (2)$$

The indexing function $\phi(i)$ is defined separately for each of two versions of Argon2: 2d and 2i. The Argon2d version, which we use, compute it as a function of the previous block $X[i-1]$.

The authors proved [10] that all the blocks are generated distinct assuming certain collision-resistant-like properties of F . They also reported the performance of 0.7 cpb on the Haswell CPU with 4 threads, and 1.6 cpb with 1 thread.

Tradeoff security of Argon2 Using the tradeoff algorithm published in [9], the authors report the values $C(\alpha)$ and $D(\alpha)$ up to $\alpha = 1/7$ with $t = 1$. It appears that $C(\alpha)$ is exponential in α , whereas $D(\alpha)$ is linear.

α	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$
$C(\alpha)$	1.5	4	20.2	344	4660	2^{18}
$D(\alpha)$	1.5	2.8	5.5	10.3	17	27

Table 1: Time and computation penalties for the ranking tradeoff attack for Argon2d.

3.2 Proofs of work

A *proof-of-work scheme* is a challenge-response protocol, where one party (Prover) has to prove (maybe probabilistically) that it has performed a certain amount of computation following a request from another party (Verifier). It typically relies on a computational problem where a solution is assumed to have fixed cost, such as the preimage search in the Bitcoin protocol and other cryptocurrencies. Other applications may include spam protection, where a proof-of-work is a certificate that is easy to produce for ordinary sender, but hard to generate in large quantities given a botnet (or more sophisticated platform).

The proof-of-work algorithm must have a few properties to be suitable for cryptocurrencies:

- It must be *amortization-free*, i.e. producing q outputs for \mathcal{B} should be q times as expensive;
- The solution must be *short* enough and verified quickly using *little memory* in order to prevent DoS attacks on the verifier.

- The time-space tradeoffs must be *steep* to prevent any price-performance reduction.
- The time and memory parameters must be *tunable independently* to sustain constant mining rate.
- To avoid a clever prover getting advantage over the others the advertised algorithm must be the most efficient algorithm to date (*optimization-freeness*).
- The algorithm must be *progress-free* to prevent centralization: the mining process must be stochastic so that the probability to find a solution grows steadily with time and is non-zero for small time periods.
- Parallelized implementations must be limited by the memory bandwidth.

As demonstrated in [11], almost any hard problem can be turned into a proof-of-work, even though it is difficult to fulfill all these properties. The well-known hard and NP-complete problems are natural candidates, since the best algorithms for them run in (sub)exponential time, whereas the verification is polynomial. The proof-of-work scheme Equihash [11] is built on the *generalized-birthday*, or k -XOR, problem, which looks for a set of n -bit strings that XOR to zero. The best existing algorithm is due to Wagner [34]. This problem is particularly interesting, as the time-space tradeoff steepness can be adjusted by changing k , which does not hold, e.g., in hard knapsacks.

Drawbacks of existing PoW We briefly discuss existing alternatives here. The first PoW schemes by Dwork and Naor [14] were just computational problems with fast verification such as the square root computation, which do not require large memory explicitly. The simplest scheme of this kind is Hashcash [7], where a partial preimage to a cryptographic hash function is found (the so called *difficulty test*). Large memory comes into play in [13], where a random array is shared between the prover and the verifier thus allowing only large-memory verifiers. This condition was relaxed in [15], where superconcentrators [28] are used to generate the array, but the verifier must still hold large memory in the initialization phase. Superconcentrators were later used in the Proof-of-Space construction [16], which allows fast verification. However, the scheme [16] if combined with the difficulty test is vulnerable to cheating (see Section 4.4 for more details) and thus can not be converted to a progress-free PoW. We note that the superconcentrators make both [15] and [16] very slow.

Ad-hoc but faster schemes started with scrypt [27], but fast verification is possible only with rather low

amount of memory. Using more memory (say, using Argon2 [10]) with a difficulty test but verifying only a subset of memory is prone to cheating as well (Section 4.4).

The scheme [11] is quite promising, but the reference implementation reported is quite slow, as it takes about 30 seconds to get a proof that certifies the memory allocation of 500 MB. As a result, the algorithm is not truly progress-free: the probability that the solution is found within the first few seconds is actually zero. It can be argued that this would stimulate centralization among the miners. In addition, the memory parameter does not have sufficient granularity and there is no correlation between the allocated memory and the minimal time needed to find the proof.

Finally, we mention schemes Momentum [21] and Cuckoo cycle [32], which provide fast verification due to their combinatorial nature. They rely on the memory requirements for the collision search (Momentum) or graph cycle finding (Cuckoo). However, Momentum is vulnerable to a sublinear time-space tradeoff [11], whereas the first version of the Cuckoo scheme was recently broken in [6].

We summarize the properties of the existing proof-of-work constructions in Table 2. The AT cost is estimated for the parameters that enable 1-second generation time on a PC.

4 MTP: Proofs of work and time-lock puzzles based on memory-hard function

In this section we present a novel proof-of-work algorithm MTP (for Merkle Tree Proof) with fast verification, which in particular solves the progress-free problem of [11]. Our approach is based on the memory-hard function, and the concrete proposal involves Argon2.

Since fast memory-hard functions \mathcal{F} such as Argon2 perform a lengthy chain of computations, but do not solve any NP-like problem, it is not fast to verify that Y is the output of F . Checking some specific (say, last) blocks does not help, as explained in detail in the further text. We thus have to design a scheme that lower bounds the time-area product for the attacker, even if he computes a slightly modified function.

4.1 Description of MTP

Consider a memory-hard function \mathcal{F} that satisfies Equation (2) (for instance, Argon2) with a single pass over the memory producing T blocks and a cryptographic hash function H (possibly used in \mathcal{F}). We propose the following non-interactive protocol for the Prover (Figure 1) in Algorithm 1, where L and d are security parameters. The average number of calls to F is $T + 2^d L$.

Algorithm 1 MTP: Merkle-tree based Proof-of-Work. Prover's algorithm

Input: Challenge I , parameters L, d .

1. Compute $\mathcal{F}(I)$ and store its T blocks $X[1], X[2], \dots, X[T]$ in the memory.
2. Compute the root Φ of the Merkle hash tree (see Appendix A).
3. Select nonce N .
4. Compute $Y_0 = H(\Phi, N)$ where G is a cryptographic hash function.
5. For $1 \leq j \leq L$:

$$i_j = Y_{j-1} \pmod{T}; \\ Y_j = H(Y_{j-1}, X[i_j]).$$

6. If Y_L has d trailing zeros, then (Φ, N, \mathcal{Z}) is the proof-of-work, where \mathcal{Z} is the opening of $2L$ blocks $\{X[i_j - 1], X[\phi(i_j)]\}$. Otherwise go to Step 3.

Output: Proof (Φ, N, \mathcal{Z}) .

The verifier, equipped with \mathcal{F} and H , runs Algorithm 2.

Algorithm 2 MTP: Verifier's algorithm

Input: Proof (Φ, N, \mathcal{Z}) , parameters L, d .

1. Compute $Y_0 = H(\Phi, N)$.
2. Verify all block openings using Φ .
3. Compute from \mathcal{Z} for $1 \leq j \leq L$:

$$X[i_j] = F(X[i_j - 1], X[\phi(i_j)]); \\ Y_j = G(Y_{j-1}, X[i_j]).$$

4. Check whether Y_L has t trailing zeros.

Output: Yes/No.

4.2 Cheating strategies

Let the computation-space tradeoff for \mathcal{H} and the default memory value T be given by functions $C(\alpha)$ and $D(\alpha)$ (Section 2).

Memory savings Suppose that a cheating prover wants to reduce the AT cost by using αT memory for some $\alpha < 1$. First, he computes $\mathcal{F}(I)$ and Φ , making $C(\alpha)T$

Scheme	AT cost	Speed	Verification		Tradeoff	Paral-sm	Progress -free
			Fast	M/less			
Dwork-Naor I [14]	Low	High	Yes	Yes	Memoryless	Yes	Yes
Dwork-Naor II [13]	High	Low	Yes	No	Memoryless	Constr.	Yes
Dwork-Naor III [15]	Medium	Low	Yes	No	Exponential	Constr.	Yes
Hashcash/Bitcoin [7]	Low	High	Yes	Yes	Memoryless	Yes	Yes
Pr.-of-Space [16]+Diff.test	High	Low	Yes	Yes	Exponential	No	No
Litecoin	Medium	High	Yes	Yes	Linear	No	Yes
Argon2-1GB + Diff.test	High	High	No	No	Exponential	No	Yes
Momentum [21]	Medium	High	Yes	Yes	Attack [11, 33]	Yes	Yes
Cuckoo cycle [32]	Medium [6]	Medium	Yes	Yes	Linear [6]	Yes	Yes
Equihash [11]	High	Medium	Yes	Yes	Exponential	Constr.	Yes
MTP	High	High	Yes	Yes	Exponential	Constr.	Yes

Table 2: Review of existing proofs of work. Litecoin utilizes scrypt with 128KB of RAM followed by the difficulty test). M/less – memoryless; constr. – constrained.

calls to F . Then for each N he has to get or recompute L blocks using only αT stored blocks. The complexity of this step is equal to the complexity of recomputing random L blocks during the first computation of \mathcal{F} . A random block is recomputed by a tree of average size $C(\alpha)$ and depth $D(\alpha)$. Therefore, to compute the proof-of-work, a memory-saving prover has to make $C(\alpha)(T + 2^d L)$ calls to F , so his amount of work grows by $C(\alpha)$.

Block modification The second cheating strategy is to compute a different function $\mathcal{F}' \neq \mathcal{F}$. More precisely, the cheater produces some blocks $X[i']$ (which we call *inconsistent* as in [16]) not as specified by Equation (2) (e.g. by simply computing $X[i'] = H(i')$). In contrast to the verifiable computation approach, our protocol allows a certain number of inconsistent blocks. Suppose that the number of inconsistent blocks is εT , then the chance that no inconsistent block is detected by L opened blocks is

$$\gamma = (1 - \varepsilon)^L.$$

Therefore, the probability for a proof-of-work with εM inconsistent blocks to pass the opening test is γ . In other words, the cheater's time is increased by the factor $1/\gamma$. We note that it does not make sense to alter the blocks after the Merkle tree computation, as any modified block would fail the opening test.

Overall cheating penalties Let us accumulate the two cheating strategies into one. Suppose that a cheater

stores αT blocks and additionally allows εT inconsistent blocks. Then he makes at least

$$\frac{C(\alpha + \varepsilon)(T + 2^d L)}{\gamma} \quad (3)$$

calls to F . The concrete values are determined by the penalty function $C()$, which depends on \mathcal{F} .

4.3 Parallelism

Both honest prover and cheater can parallelize the computation for 2^t different nonces. However, the latency of cheater's computation will be higher, since each block generates a recomputation tree of average depth $D(\alpha + \varepsilon)$.

4.4 Why simpler approach does not work: grinding attack

Now we can explain in more details why the composition of \mathcal{F} and the difficulty test is not a good proof-of-work even if some internal blocks of \mathcal{H} are opened. Suppose that the proof is accepted if $H(X[T])$ has certain number d of trailing zeros. One would expect that a prover has to try 2^d distinct I on average and thus call \mathcal{F} 2^t times to find a solution. However, a cheating prover can simply try 2^d values for $X[T]$ and find one that passes the test in just 2^d calls to H . Although $X[T]$ is now inconsistent, it is unlikely to be selected among L blocks to open, so the cheater escapes detection easily. Additionally checking $X[T]$ would not resolve the problem since a cheater

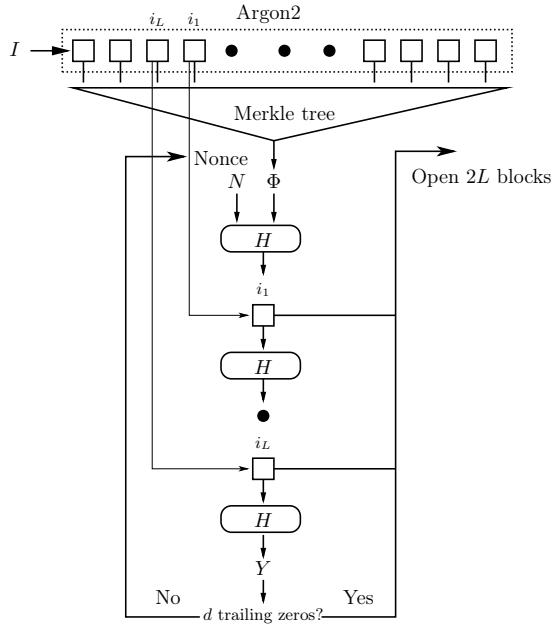


Figure 1: MTP: Merkle-tree based Proof-of-Work with light verification.

would then modify the previous block, or $X[\phi(T)]$, or an earlier block and then propagate the changes. A single inconsistent block is just too difficult to catch².

4.5 MTP-Argon2

As a concrete application, we suggest a cryptocurrency proof-of-work based on Argon2d with 4 parallel lanes. We aim to make this PoW unattractive for botnets, so we suggest using 2 GB of RAM, which is very noticeable (and thus would likely alarm the user), while being bearable for the regular user, who consciously decided to use his desktop for mining. On our 1.8 GHz machine a single call to 2-GB Argon2d runs in 0.5 seconds, but the Merkle tree computation is more expensive, as we have to hash 2 GB of data splitted into 1 KB blocks. We suggest using Blake2b for H , as it is already used in Argon2d, but restrict to 128-bit output, so that the total running time is about 3 seconds. In this case a single opening has $16 \cdot 21$ bytes of hashes, or 1.3 KB in total.

We suggest $L = 70$, so that the entire proof consists of 140 blocks and their openings, or 180 KB in total. Let us figure out the cheating advantage. The $C()$ and $D()$ functions are given in Table 1). Assuming certain ratio between the area needed to implement Blake2b and

²We have not seen any formal treatment of this attack in the literature, but it appears to be known in the community. It is mentioned in [26] and [4].

the area needed for DRAM, we get the following lower bound on the ASIC-equipped cheater.

Proposition 1 For $L = 70$ and 2 GB of RAM the time-area product can be reduced by the factor of 12 at most, assuming that each Blake2b core occupies an equivalent of 2^{16} bytes.

Proof. Assuming that each core occupies 2^{16} bytes, we obtain $\beta = 2^{-15}$ in terms of Equation (1). Since the cheater has the success chance $\gamma = (1 - \varepsilon)^L$, Equation (1) is modified as follows:

$$AT_\alpha = AT_1 \frac{\alpha D(\alpha + \varepsilon) + C(\alpha + \varepsilon)/2^{15}}{(1 - \varepsilon)^L}. \quad (4)$$

Consider three options:

- $\alpha, \varepsilon < 1/12$. Then $C(\alpha + \varepsilon) \geq 4660$ (Table 1) and we have

$$AT_\alpha \geq AT_1 \cdot \frac{4660}{32768} \geq AT_1 \cdot 0.12.$$

- $\alpha < 1/12, 1/6 \geq \varepsilon > 1/12$. Then $C(\alpha + \varepsilon) \geq 20$, $(1 - \varepsilon)^L > 1/441$, and we have

$$AT_\alpha \geq AT_1 \cdot \frac{20 \cdot 441}{32768} \geq AT_1 \cdot 0.27.$$

- $\alpha < 1/12, 1/6 \leq \varepsilon$. Then $(1 - \varepsilon)^L > 2^{15}$, and the time-area product increases.
- $\alpha > 1/12$. Then $AT_\alpha \geq AT_1 \cdot 1/12$.

This ends the proof.

We conclude that a cheater can gain at most 12x-advantage, whereas he can still be detected in the future by memory-rich verifiers. Tradeoffs are also not helpful when implementing this Proof-of-Work on ASIC. Altogether, our proposal should reduce the relative efficiency of potential ASIC mining rigs and allow more egalitarian mining process. Even if someone decides to use large botnets (10,000 machines and more), all the botnets machines would have to use the same 2 GB of memory, otherwise they would suffer large penalty. We note that if $\varepsilon = 0$, i.e. the prover is honest, then his maximal advantage is $\max \frac{1}{\alpha D(\alpha)} \leq 2$.

4.6 MTP as a tool for time-lock puzzles and timestamping

The paradigm of inherently sequential computation was developed by [12] in the application to CPU benchmarking and [29] for timestamping, i.e. to certify that the document was generated certain amount of time in the past. Rivest et al. suggested *time-lock puzzles* for this purpose.

In our context, a time-lock puzzle solution is a proof-of-work that has lower bound on the running time assuming unlimited parallelism.

The verifier in [20, 29] selects a prime product $N = pq$ and asks the prover to compute the exponent $2^{2^D} \pmod{N}$ for some $D \approx N$. It is conjectured that the prover who does not know the factors can not exponentiate faster than do D consecutive squarings. In turn, the verifier can verify the solution by computing the exponent 2^D modulo $\phi(N)$, which takes $\log(D)$ time. So far the conjecture has not been refuted, but the scheme inherently requires a secret held by the verifier, and thus is not suitable for proofs-of-work without secrets, as in cryptocurrencies.

Time-lock puzzles without secrets were suggested by Mahmoody et al. [22]. Their construction is a graph of hash computations, which is based on depth-robust graphs similarly to [16]. The puzzle is a deterministic graph such that removing any constant fraction of nodes keeps its depth above the constant fraction of the original one (so the parallel computation time is lower bounded). A Merkle tree is put atop of it with its root determining a small number of nodes to open. Therefore, a cheater who wants to compute the graph in less time has to subvert too many nodes and is likely to be caught. As [16], the construction by Mahmoody et al., if combined with the difficulty filter, is subject to the grinding attack described above.

The MTP-Argon2 construction can be viewed as a time-lock puzzle and an improvement over these schemes. First, the difficulty filter is explicitly based on the grinding attack, which makes it a legitimate way to solve the puzzle. Secondly, it is much faster due to high speed of Argon2d. The time-lock property comes from the fact that the computation chain can not be parallelized as the graph structure is not known before the computation.

Suppose that MTP-Argon2 is parallelized by the additional factor of R so that each core computes a chain of length about T/R . Let core j compute j -th (out of R) chain, chronologically. Then at step i each core has computed i blocks and has not computed $T/R - i$ blocks, so the probability that core j requests a block that has not been computed is

$$\frac{(j-1)(T/R-i)}{(j-1)T/R+i} \leq \frac{(j-1)(T/R-i)}{jT/R}.$$

Summing by all i , we obtain that core j misses at least $\frac{T(1-1/j)}{2R}$, so the total fraction of inconsistent blocks is about $0.5 - \frac{\ln R}{2R}$. Therefore, ϵ quickly approaches 0.5, which is easily detectable. We thus conclude that a parallel implementation of MTP-Argon2 is likely to fail the Merkle tree verification.

5 Memory-hard encryption on low-entropy keys

5.1 Motivation

In this section we approach standard encryption from the memory-hardness perspective. A typical approach to hard-drive encryption is to derive the master key from the user password and then use it to encrypt chunks of data in a certain mode of operation such as XTS [24]. The major threat, as to other password-based security schemes, are low-entropy passwords. An attacker, who gets access to the hard drive encrypted with such password, can determine the correct key and then decrypt within short time.

A countermeasure could be to use a memory-hard function for the key derivation, so that the trial keys can be produced only on memory-rich machines. However, the trial decryption could still be performed on special memoryless hardware given these keys. We suggest a more robust scheme which covers this type of adversaries and eventually requires that the entire attack code have permanent access to large memory.

5.2 Requirements

We assume the following setting, which is inspired by typical disk-encryption applications. The data consists of multiple chunks $Q \in \mathcal{Q}$, which can be encrypted and decrypted independently. The only secret that is available to the encryption scheme \mathcal{E} is the user-input password $P \in \mathcal{P}$, which has sufficiently low entropy to be memorized (e.g., 6 lowercase symbols). The encryption syntax is then as follows:

$$\mathcal{E} : \mathcal{P} \times \mathcal{S} \times \mathcal{Q} \rightarrow \mathcal{C},$$

where $S \in \mathcal{S}$ is associated data, which may contain salt, encryption nonce or IV, chunk identifier, time, and other secondary input; and $C \in \mathcal{C}$ is ciphertext. S serves both to simplify ciphertext identification (as it is public) and to ensure certain cryptographic properties. For instance, unique salt or nonce prevents repetition of ciphertexts for identical plaintexts. We note that in some settings due to storage restriction the latter requirement can be dropped. Decryption then is naturally defined and we omit its formal syntax.

In our proposal we do not restrict the chunk size. Even though it can be defined for chunks as small as disk sectors, the resistance to cracking attacks will be higher for larger chunks, up to a megabyte long.

A typical attack setting is as follows. An attacker obtains the encrypted data via some malicious channel or installs malware and then tries different passwords to decrypt it. For the sake of simplicity, we assume that the

plaintext contains sufficient redundancy so that a successful guess can be identified easily. Therefore, the adversary tries D passwords from his dictionary $\mathcal{D} \subset \mathcal{P}$. Let T be the time needed for the fastest decryption operation that provides partial knowledge of plaintext sufficient to discard or remember the password, and A_0 be the chip area needed to implement this operation. Then the total amount of work performed by the adversary is

$$W = D \cdot T \cdot A_0.$$

At the same time, the time to encrypt T' for a typical user should not be far larger than T . Our goal is to maximize W with keeping T' the same or smaller.

The memory-hard functions seem to serve perfectly for the purpose of maximizing W . However, it remains unclear how to combine such function \mathcal{F} with \mathcal{E} to get *memory-hard encryption* (MHE).

Now we formulate some additional features that should be desirable for such a scheme:

- The user should be able to choose the requested memory size A independently of the chunk length $|Q|$. Whereas the chunk length can be primarily determined by the CPU cache size, desirable processing speed, or the hard drive properties, the memory size determines the scheme’s resistance to cracking attacks.
- The memory can be allocated independently for each chunk or reused. In the former case the user can not allocate too much memory as the massive decryption would be too expensive. However, for the amounts of memory comparable to the chunk size the memory-hard decryption should take roughly as much as memoryless decryption. If the allocated memory is reused for distinct chunks, much more memory can be allocated as the allocation time can be amortized. However, the decryption latency would be quite high. We present both options in the further text.
- Full ciphertext must be processed to decrypt a single byte. This property clearly makes T larger since the adversary would have to process an entire chunk to check the password. At the same time, for disk encryption it should be fine to decrypt in the “all-or-nothing” fashion, as the decryption time would still be smaller than the user could wait.
- Encryption should be done in one pass over data. It might sound desirable that the decryption should be done in one pass too. However, this would contradict the previous requirement. Indeed, if the decryption can be done in one pass, then the first bytes

of the plaintext can be determined without the last bytes of the ciphertext³.

- Apart from the memory parameter, the total time needed to allocate this memory should be tunable too. It might happen that the application does not have sufficient memory but does have time. In this case, the adversary can be slowed down by making several passes over the memory during its initialization (the memory-hard function that we consider support this feature).

Our next and final requirement comes from adversary’s side. When the malware is used, the incoming network connection and memory for this malware can be limited. Thus, it would be ideal for the attacker if the memory-intensive part can be delegated to large machines under attacker’s control, such as botnets. If we just derived the secret-key K for encryption as the output of the memory-hard hash function \mathcal{F} , this would be exactly this case. An adversary would then run \mathcal{F} for dictionary D on his own machine, produce the set \mathcal{K} of keys, and supply them to malware (recall that due to low entropy there would be only a handful of these keys). Thus the final requirement should be the following:

- During decryption, it should be impossible to delegate the entire memory-hard computation to the external device without accessing the ciphertext. Therefore, there could be no memory-hard precomputation.

5.3 Our scheme

Our scheme is based on a recent proposal by Zaverucha [35], who addresses similar properties in the scheme based on Rivest’s All-or-Nothing transform (ANT). However, the scheme in [35] does not use an external memory-hard function, which makes it memory requirements inevitably bound to the chunk size. Small chunks but large memory is impossible in [35].

Our proposal is again based on the All-or-Nothing transformation, though we expect that similar properties can be obtained with deterministic authenticated encryption scheme as a core primitive. The chunk length q (measured in blocks using by \mathcal{F}) and memory size $M \geq q$ are the parameters as well as some blockcipher E (possibly AES). First, we outline the scheme where the memory is allocated separately for each chunk. The reader may also refer to Figure 2.

The underlying idea is to use both the header and the body blocks to produce the ciphertext. In turn, to recompute the body blocks both the ciphertext and the header must be available during trial decryption.

³The similar argument is made for the online authenticated ciphers

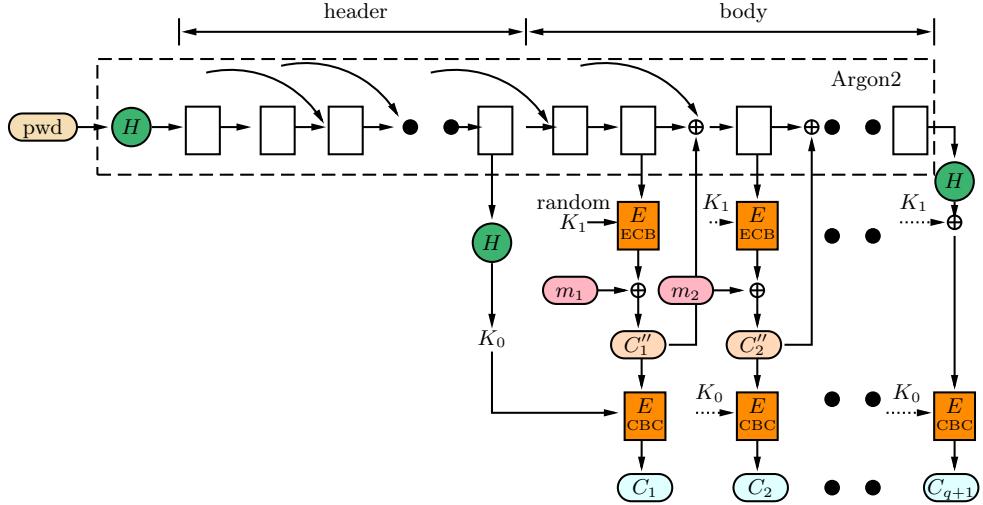


Figure 2: MHE: Disk encryption using memory-hard function Argon2.

The version of the MHE scheme which allocates the same memory for multiple chunks is very similar. The S input is ignored at the beginning, so that the header memory blocks do not depend on the data. Instead, we set $K_0 = H(X_0, S)$, so that the body blocks are affected by S and M , and thus are different for every chunk. In this case the body blocks have to be stored separately and should not overwrite the header blocks for $t > 1$.

Let us verify that the scheme in Algorithm 3 satisfies the properties we listed earlier:

- The allocated memory size M can be chosen independently of the chunk length q (as long as $M > q$).
- The body memory blocks are allocated and processed for each chunk independently. In addition, the header blocks are also processed independently for each chunk in the single-chunk version.
- In order to decrypt a single byte of the ciphertext, an adversary would have to obtain K_1 , which can be done only by running \mathcal{F} up to the final block, which requires all C''_i , which are in turn must be derived from the ciphertext blocks.
- Encryption needs one pass over data, and decryption needs two passes over data.
- The total time needed to allocate and fill the header is tunable.
- The computation of the body memory blocks during decryption can not be delegated, as it requires knowledge both of the header and the ciphertext. It

in [18].

might be possible to generate the header on an external machine, but then random access to its blocks to decrypt the ciphertext is required.

We note that properties 1, 5, and 6 are not present in [35].

Security First, we address traditional CPA security. We do not outline the full proof here, just the basic steps. We assume that the adversary does not have access to the internals of Argon2, and that blockcipher E is a secure PRF. Next, we assume collision-resistance of the compression function F used in \mathcal{F} . Given that, we prove that all the memory blocks are distinct, which yields the CPA security for C' . From the latter we deduce the CPA security for the final ciphertext. We note that in the case when the collision-resistance of F can not be guaranteed, we may additionally require that X_i undergo hashing by a cryptographic hash function H' before encryption, so that the plaintext blocks are still distinct. All these properties hold up to the birthday bound of the blockcipher.

Next, we figure out the tradeoff security. The genuine decrypting user is supposed to spend M memory blocks for \mathcal{F} and q memory blocks to store the plaintext and intermediate variables (if the ciphertext can be overwritten, then these q blocks are not needed). Suppose that an adversary wants to use αM memory for header and body. Then each missing block, if asked during decryption, must be recomputed making $C(\alpha)$ calls to F . The best such strategy for Argon2, described in [9], yields $C(\alpha)$ that grows exponentially in $1/\alpha$. For example, using $1/5$ of memory, an adversary would have to make 344 times as many calls to F , which makes a memory-reducing encryption cracking inefficient even on special hardware.

Algorithm 3 Memory-hard encryption with independent memory allocation (for each chunk).

Input: Password P , memory size M , associated data S , chunk Q , number of iterations t , memory-hard function \mathcal{F} (preferably Argon2), blockcipher E , cryptographic hash function H (e.g. SHA-3).

1. Run \mathcal{F} on (P, S) with input parameters M and t but fill only $M - q$ blocks (the *header*) in the last iteration. Let X_0 be the last memory block produced by \mathcal{F} .
2. Produce $K_0 = H(X_0)$ — the first session key.
3. Generate a random session key K_1 .
4. Generate the remaining blocks X_1, X_2, \dots, X_q (*body*) for \mathcal{F} as follows. We assume that each chunk M consists of smaller blocks m_1, m_2, \dots, m_q of length equal to the block size of \mathcal{F} . For each $i \geq 1$:
 - Encrypt X_{i-1} by E in the ECB mode under K_1 and get the intermediate ciphertext block C'_i .
 - Add the chunk data: $C''_i = C'_i \oplus m_i$.
 - Encrypt C''_i under K_0 in the CBC mode and produce the final ciphertext block C_i .
 - Modify the memory: $X_{i-1} \leftarrow X_{i-1} \oplus C''_i$.
 - Generate the block X_i according to the specification of \mathcal{F} . In Argon2, the modified X_{i-1} and some another block $X[\phi(X_{i-1})]$ would be used.
5. After the entire chunk is encrypted, encrypt also the key K_1 :

$$C_{t+1} = E_{K_0}(H(X_t) \oplus K_1).$$

Output: C_1, \dots, C_{t+1} .

Performance We suggest taking $l = 4$ in Argon2 in order to fill the header faster using multiple cores, which reportedly takes 0.7 cpb (about the speed of AES-GCM and AES-XTS). The body has to be filled sequentially as the encryption process is sequential. As AES-CBC is about 1.3 cpb, and we use two of it, the body phase should run at about 4 cpb. In a concrete setting, suppose that we tolerate 0.1 second decryption time (about 300 Mcycles) for the 1-MB chunk. Then we can take the header as large as 256 MB, as it would be processed in 170 Mcycles + 4 Mcycles for the body phase.

6 Conclusion

We have introduced the new paradigm of egalitarian computing, which suggests amalgamating arbitrary computation with a memory-hard function to enhance the security against off-line adversaries equipped with powerful tools (in particular with optimized hardware). We have reviewed password hashing and proofs of work as applications where such schemes are already in use or are planned to be used. We then introduce two more schemes in this framework. The first one is MTP, the progress-free proof-of-work scheme with fast verification based on the memory-hard function Argon2, the winner of the Password Hashing Competition. The second scheme pioneers the memory-hard encryption — the security enhancement for password-based disk encryption, also based on Argon2.

References

- [1] Avalonasic's 40nm chip to bring hashing boost for less power, 2014. <http://www.coindesk.com/avalon-asics-40nm-/chip-brings-hashing-boost-less-power/>.
- [2] Bitcoin: Mining hardware comparison, 2014. available at https://en.bitcoin.it/wiki/Mining_hardware_comparison. We compare 2^{32} hashes per joule on the best ASICs with 2^{17} hashes per joule on the most efficient x86-laptops.
- [3] Password Hashing Competition, 2015. <https://password-hashing.net/>.
- [4] 2016. Andrew Miller, Bram Cohen, private communication.
- [5] ABADI, M., BURROWS, M., AND WOBBER, T. Moderately hard, memory-bound functions. In NDSS'03 (2003), The Internet Society.
- [6] ANDERSEN, D. A public review of cuckoo cycle. <http://www.cs.cmu.edu/~dga/crypto/cuckoo/analysis.pdf>, 2014.
- [7] BACK, A. Hashcash – a denial of service counter-measure, 2002. available at <http://www.hashcash.org/papers/hashcash.pdf>.
- [8] BERNSTEIN, D. J., AND LANGE, T. Non-uniform cracks in the concrete: The power of free precomputation. In ASIACRYPT'13 (2013), vol. 8270 of *Lecture Notes in Computer Science*, Springer, pp. 321–340.

- [9] BIRYUKOV, A., AND KHOVRATOVICH, D. Tradeoff cryptanalysis of memory-hard functions. In *Asiacrypt’15* (2015). available at <http://eprint.iacr.org/2015/227>.
- [10] BIRYUKOV, A., AND KHOVRATOVICH, D. Argon2: new generation of memory-hard functions for password hashing and other applications. In *Euro S&P’16* (2016). available at <https://www.cryptolux.org/images/0/0d/Argon2.pdf>.
- [11] BIRYUKOV, A., AND KHOVRATOVICH, D. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. In *NDSS’16* (2016). available at <https://eprint.iacr.org/2015/946.pdf>.
- [12] CAI, J., LIPTON, R. J., SEDGEWICK, R., AND YAO, A. C. Towards uncheatable benchmarks. In *Structure in Complexity Theory Conference* (1993), IEEE Computer Society, pp. 2–11.
- [13] DWORK, C., GOLDBERG, A., AND NAOR, M. On memory-bound functions for fighting spam. In *CRYPTO’03* (2003), vol. 2729 of *Lecture Notes in Computer Science*, Springer, pp. 426–444.
- [14] DWORK, C., AND NAOR, M. Pricing via processing or combatting junk mail. In *CRYPTO’92* (1992), vol. 740 of *Lecture Notes in Computer Science*, Springer, pp. 139–147.
- [15] DWORK, C., NAOR, M., AND WEE, H. Pebbling and proofs of work. In *CRYPTO’05* (2005), vol. 3621 of *Lecture Notes in Computer Science*, Springer, pp. 37–54.
- [16] DZIEMBOWSKI, S., FAUST, S., KOLMOGOROV, V., AND PIETRZAK, K. Proofs of space. In *CRYPTO’15* (2015), R. Gennaro and M. Robshaw, Eds., vol. 9216 of *Lecture Notes in Computer Science*, Springer, pp. 585–605.
- [17] GIRIDHAR, B., CIESLAK, M., DUGGAL, D., DRESLINSKI, R. G., CHEN, H., PATTI, R., HOLD, B., CHAKRABARTI, C., MUDGE, T. N., AND BLAAUW, D. Exploring DRAM organizations for energy-efficient and resilient exascale memories. In *International Conference for High Performance Computing, Networking, Storage and Analysis 2013* (2013), ACM, pp. 23–35.
- [18] HOANG, V. T., REYHANITABAR, R., ROGAWAY, P., AND VIZÁR, D. Online authenticated-encryption and its nonce-reuse misuse-resistance. In *CRYPTO’15* (2015), R. Gennaro and M. Robshaw, Eds., vol. 9215 of *Lecture Notes in Computer Science*, Springer, pp. 493–517.
- [19] HOPCROFT, J. E., PAUL, W. J., AND VALIANT, L. G. On time versus space. *J. ACM* 24, 2 (1977), 332–337.
- [20] JERSCHOW, Y. I., AND MAUVE, M. Offline submission with RSA time-lock puzzles. In *CIT* (2010), IEEE Computer Society, pp. 1058–1064.
- [21] LORIMER, D. Momentum – a memory-hard proof-of-work via finding birthday collisions, 2014. available at <http://www.hashcash.org/papers/momentum.pdf>.
- [22] MAHMOODY, M., MORAN, T., AND VADHAN, S. P. Publicly verifiable proofs of sequential work. In *ITCS* (2013), ACM, pp. 373–388.
- [23] MALVONI, K. Energy-efficient bcrypt cracking, 2014. Passwords’14 conference, available at <http://www.openwall.com/presentations/Passwords14-Energy-Efficient-Cracking/>.
- [24] MARTIN, L. Xts: A mode of aes for encrypting hard disks. *IEEE Security & Privacy*, 3 (2010), 68–69.
- [25] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. <http://www.bitcoin.org/bitcoin.pdf>.
- [26] PARK, S., PIETRZAK, K., ALWEN, J., FUCHSBAUER, G., AND GAZI, P. Spacecoin: A cryptocurrency based on proofs of space. *IACR Cryptology ePrint Archive* 2015 (2015), 528.
- [27] PERCIVAL, C. Stronger key derivation via sequential memory-hard functions. <http://www.tarsnap.com/scrypt/scrypt.pdf>.
- [28] PIPPENGER, N. Superconcentrators. *SIAM J. Comput.* 6, 2 (1977), 298–304.
- [29] RIVEST, R. L., SHAMIR, A., AND WAGNER, D. A. Time-lock puzzles and timed-release crypto. <https://people.csail.mit.edu/rivest/pubs/RSW96.pdf>.
- [30] SPRENGERS, M., AND BATINA, L. Speeding up GPU-based password cracking. In *SHARCS’12* (2012). available at <http://2012.sharcs.org/record.pdf>.
- [31] THOMPSON, C. D. Area-time complexity for VLSI. In *STOC’79* (1979), ACM, pp. 81–88.
- [32] TROMP, J. Cuckoo cycle: a memory bound graph-theoretic proof-of-work. Cryptology ePrint Archive, Report 2014/059, 2014. available at <http://eprint.iacr.org/2014/059>, project webpage <https://github.com/tromp/cuckoo>.
- [33] VAN OORSCHOT, P. C., AND WIENER, M. J. Parallel collision search with cryptanalytic applications. *J. Cryptology* 12, 1 (1999), 1–28.
- [34] WAGNER, D. A generalized birthday problem. In *CRYPTO’02* (2002), vol. 2442 of *Lecture Notes in Computer Science*, Springer, pp. 288–303.
- [35] ZAVERUCHA, G. Stronger password-based encryption using all-or-nothing transforms. available at <http://research.microsoft.com/pubs/252097/pbe.pdf>.

A Merkle hash trees

We use Merkle hash trees in the following form. A prover P commits to T blocks $X[1], X[2], \dots, X[T]$ by computing the hash tree where the blocks $X[i]$ are at leaves at depth $\log T$ and nodes compute hashes of their branches. For instance, for $T = 4$ and hash function G prover P computes and publishes

$$\Phi = G(G(X[1], X[2]), G(X[3], X[4])).$$

Prover stores all blocks and all intermediate hashes. In order to prove that he knows, say, $X[5]$ for $T = 8$, (or to *open* it) he discloses the hashes needed to reconstruct the path from $X[5]$ to Φ :

$$\begin{aligned} \text{open}(X[5]) &= (X[5], X[6], g_{78} = G(X[7], X[8])), \\ &\quad g_{1234} = G(G(X[1], X[2]), G(X[3], X[4])), \Phi), \end{aligned}$$

so that the verifier can make all the computations. If G is collision-resistant, it is hard to open any block in more than one possible way.

Post-quantum key exchange – a new hope*

Erdem Alkim

Department of Mathematics, Ege University, Turkey

Léo Ducas

Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands

Thomas Pöppelmann

Infineon Technologies AG, Munich, Germany

Peter Schwabe

Digital Security Group, Radboud University, The Netherlands

Abstract

At IEEE Security & Privacy 2015, Bos, Costello, Naehrig, and Stebila proposed an instantiation of Peikert’s ring-learning-with-errors-based (Ring-LWE) key-exchange protocol (PQCrypto 2014), together with an implementation integrated into OpenSSL, with the affirmed goal of providing post-quantum security for TLS. In this work we revisit their instantiation and stand-alone implementation. Specifically, we propose new parameters and a better suited error distribution, analyze the scheme’s hardness against attacks by quantum computers in a conservative way, introduce a new and more efficient error-reconciliation mechanism, and propose a defense against backdoors and all-for-the-price-of-one attacks. By these measures and for the same lattice dimension, we more than double the security parameter, halve the communication overhead, and speed up computation by more than a factor of 8 in a portable C implementation and by more than a factor of 27 in an optimized implementation targeting current Intel CPUs. These speedups are achieved with comprehensive protection against timing attacks.

1 Introduction

The last decade in cryptography has seen the birth of numerous constructions of cryptosystems based on lattice problems, achieving functionalities that were previously unreachable (e.g., fully homomorphic cryptogra-

phy [38]). But even for the simplest tasks in asymmetric cryptography, namely public-key encryption, signatures, and key exchange, lattice-based cryptography offers an important feature: resistance to all known quantum algorithms. In those times of *quantum nervousness* [73, 74], the time has come for the community to deliver and optimize concrete schemes, and to get involved in the standardization of a lattice-based cipher-suite via an open process.

For encryption and signatures, several competitive schemes have been proposed; examples are NTRU encryption [50, 83], Ring-LWE encryption [67] as well as the signature schemes BLISS [31], PASS [48] or the proposal by Bai and Galbraith presented in [8]. To complete the lattice-based cipher-suite, Bos et al. [20] recently proposed a concrete instantiation of the key-exchange scheme of Peikert’s improved version of the original protocol of Ding, Xie and Lin [52, 77]. Bos et al. proved its practicality by integrating their implementation as additional cipher-suite into the transport layer security (TLS) protocol in OpenSSL. In the following we will refer to this proposal as BCNS.

Unfortunately, the performance of BCNS seemed rather disappointing. We identify two main sources for this inefficiency. First the analysis of the failure probability was far from tight, resulting in a very large modulus $q \approx 2^{32}$. As a side effect, the security is also significantly lower than what one could achieve with Ring-LWE for a ring of rank $n = 1024$. Second the Gaussian sampler, used to generate the secret parameters, is fairly inefficient and hard to protect against timing attacks. This second source of inefficiency stems from the fundamental misconception that high-quality Gaussian noise is crucial for encryption based on LWE¹, which has also made various other implementations [29, 79] slower and more complex than they would have to be.

*This work was initiated while Thomas Pöppelmann was a Ph.D. student at Ruhr-University Bochum with support from the European Union H2020 SAFCrypto project (grant no. 644729). This work has furthermore been supported by TÜBITAK under 2214-A Doctoral Research Program Grant, by the European Commission through the ICT program under contract ICT-645622 (PQCRIPTO), and by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114 and through a Free Competition Grant. Permanent ID of this document: 0462d84a3d34b12b75e8f5e4ca032869. Date: 2016-06-28.

¹This is very different for lattice-based signatures or trapdoors, where distributions need to be meticulously crafted to prevent any leak of information on a secret basis.

1.1 Contributions

In this work, we propose solutions to the performance and security issues of the aforementioned BCNS proposal [20]. Our improvements are possible through a combination of multiple contributions:

- Our first contribution is an improved analysis of the failure probability of the protocol. To push the scheme even further, inspired by analog error-correcting codes, we make use of the lattice D_4 to allow error reconciliation beyond the original bounds of [77]. This drastically decreases the modulus to $q = 12289 < 2^{14}$, which improves both efficiency and security.
- Our second contribution is a more detailed security analysis against quantum attacks. We provide a lower bound on all known (or even pre-supposed) quantum algorithms solving the shortest-vector problem (SVP), and deduce the potential performance of a quantum BKZ algorithm. According to this analysis, our improved proposal provides 128 bits of post-quantum security with a comfortable margin.
- We furthermore propose to replace the almost-perfect discrete Gaussian distribution by something relatively close, but much easier to sample, and prove that this can only affect the security marginally.
- We replace the fixed parameter \mathbf{a} of the original scheme by a freshly chosen random one in each key exchange. This incurs an acceptable overhead but prevents backdoors embedded in the choice of this parameter and all-for-the-price-of-one attacks.
- We specify an encoding of polynomials in the number-theoretic transform (NTT) domain which allows us to eliminate some of the NTT transformations inside the protocol computation.
- To demonstrate the applicability and performance of our design we provide a portable reference implementation written in C and a highly optimized vectorized implementation that targets recent Intel CPUs and is compatible with recent AMD CPUs. We describe an efficient approach to lazy reduction inside the NTT, which is based on a combination of Montgomery reductions and short Barrett reductions.

Availability of software. We place all software described in this paper into the public domain and make it available online at <https://cryptojedi.org/papers/#newhope>.

org/crypto/#newhope and <https://github.com/troeppelmann/newhope>.

Full version of the paper. The full version of this paper contains various appendices in addition to the material presented in this proceedings version. The full version is available online at <https://eprint.iacr.org/2015/1092/> and at <https://cryptojedi.org/papers/#newhope>.

Acknowledgments. We are thankful to Mike Hamburg and to Paul Crowley for pointing out mistakes in a previous version of this paper, and we are thankful to Isis Lovecraft for thoroughly proofreading the paper and for suggesting the name JARJAR for the low-security variant of our proposal.

2 Lattice-based key exchange

Let \mathbb{Z} be the ring of rational integers. We define for an $x \in \mathbb{R}$ the rounding function $\lfloor x \rfloor = \lfloor x + \frac{1}{2} \rfloor \in \mathbb{Z}$. Let \mathbb{Z}_q , for an integer $q \geq 1$, denote the quotient ring $\mathbb{Z}/q\mathbb{Z}$. We define $\mathcal{R} = \mathbb{Z}[X]/(X^n + 1)$ as the ring of integer polynomials modulo $X^n + 1$. By $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ we mean the ring of integer polynomials modulo $X^n + 1$ where each coefficient is reduced modulo q . In case χ is a probability distribution over \mathcal{R} , then $x \xleftarrow{\$} \chi$ means the sampling of $x \in \mathcal{R}$ according to χ . When we write $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$ this means that all coefficients of \mathbf{a} are chosen uniformly at random from \mathbb{Z}_q . For a probabilistic algorithm \mathcal{A} we denote by $y \xleftarrow{\$} \mathcal{A}$ that the output of \mathcal{A} is assigned to y and that \mathcal{A} is running with randomly chosen coins. We recall the discrete Gaussian distribution $D_{\mathbb{Z}, \sigma}$ which is parametrized by the Gaussian parameter $\sigma \in \mathbb{R}$ and defined by assigning a weight proportional to $\exp(\frac{-x^2}{2\sigma^2})$ to all integers x .

2.1 The scheme of Peikert

In this section we briefly revisit the passively secure key-encapsulation mechanism (KEM) that was proposed by Peikert [77] and instantiated in [20] (BCNS). Peikert’s KEM scheme is defined by the algorithms (Setup, Gen, Encaps, Decaps) and after a successful protocol run both parties share an ephemeral secret key that can be used to protect further communication (see Protocol 1).

The KEM scheme by Peikert closely resembles a previously introduced Ring-LWE encryption scheme [66] but due to a new error-reconciliation mechanism, one \mathcal{R}_q component of the ciphertext can be replaced by a more compact element in \mathcal{R}_2 . This efficiency gain is possible due to the observation that it is not necessary to transmit an explicitly chosen key to establish a secure

ephemeral session key. In Peikert’s scheme, the reconciliation just allows both parties to derive the session key from an approximately agreed pseudorandom ring element. For Alice, this ring element is $\mathbf{us} = \mathbf{ass}' + \mathbf{e}'$ s and for Bob it is $\mathbf{v} = \mathbf{bs}' + \mathbf{e}'' = \mathbf{ass}' + \mathbf{es}' + \mathbf{e}''$. For a full explanation of the reconciliation we refer to the original paper [77] but briefly recall the cross-rounding function $\langle \cdot \rangle_2$ defined as $\langle v \rangle_2 := \lfloor \frac{4}{q} \cdot v \rfloor \pmod{2}$ and the randomized function $\text{dbl}(v) := 2v - \bar{e}$ for some random \bar{e} where $\bar{e} = 0$ with probability $\frac{1}{2}$, $\bar{e} = 1$ with probability $\frac{1}{4}$, and $\bar{e} = -1$ with probability $\frac{1}{4}$. Let $I_0 = \{0, 1, \dots, \lfloor \frac{q}{2} \rfloor - 1\}$, $I_1 = \{-\lfloor \frac{q}{2} \rfloor, \dots, -1\}$, and $E = [-\frac{q}{4}, \frac{q}{4}]$ then the reconciliation function $\text{rec}(w, b)$ is defined as

$$\text{rec}(w, b) = \begin{cases} 0, & \text{if } w \in I_b + E \pmod{q} \\ 1, & \text{otherwise.} \end{cases}$$

If these functions are applied to polynomials this means they are applied to each of the coefficients separately.

Parameters: q, n, χ	
KEM. $\text{Setup}()$:	
$\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$	
Alice (server)	Bob (client)
KEM. $\text{Gen}(\mathbf{a})$:	KEM. $\text{Encaps}(\mathbf{a}, \mathbf{b})$:
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$	$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{\mathbf{b}}$ $\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$ $\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$ $\bar{\mathbf{v}} \xleftarrow{\$} \text{dbl}(\mathbf{v})$
KEM. $\text{Decaps}(\mathbf{s}, (\mathbf{u}, \mathbf{v}'))$:	$\xleftarrow{\mathbf{u}, \mathbf{v}'}$ $\mu \leftarrow \text{rec}(2\mathbf{us}, \mathbf{v}')$ $\mu \leftarrow \lfloor \bar{\mathbf{v}} \rfloor_2$

Protocol 1: Peikert’s KEM mechanism.

2.2 The BCNS proposal

In a work by Bos, Costello, Naehrig, and Stebila [20] (BCNS), Peikert’s KEM [77] was phrased as a key-exchange protocol (see again Protocol 1), instantiated for a concrete parameter set, and integrated into OpenSSL (see Section 8 for a performance comparison). Selection of parameters was necessary as Peikert’s original work does not contain concrete parameters and the security as well as error estimation are based on asymptotics. The authors of [20] chose a dimension $n = 1024$, a modulus $q = 2^{32} - 1$, $\chi = D_{\mathbb{Z}, \sigma}$ and the Gaussian parameter $\sigma = 8/\sqrt{2\pi} \approx 3.192$. It is claimed that these parameters provide a classical security level of at least 128 bits considering the distinguishing attack [62] with distinguishing advantage less than 2^{-128} and $2^{81.9}$ bits of security

against an optimistic instantiation of a quantum adversary. The probability of a wrong key being established is less than $2^{-2^{17}} = 2^{-131072}$. The message \mathbf{b} sent by Alice is a ring element and thus requires at least $\log_2(q)n = 32$ kbits while Bob’s response (\mathbf{u}, \mathbf{r}) is a ring element R_q and an element from R_2 and thus requires at least 33 kbits. As the polynomial $\mathbf{a} \in \mathcal{R}_q$ is shared between all parties this ring element has to be stored or generated on-the-fly. For timings of their implementation we refer to Table 2. We would also like to note that besides its aim for securing classical TLS, the BCNS protocol has already been proposed as a building block for Tor [84] on top of existing elliptic-curve infrastructure [41].

2.3 Our proposal: NEWHOPE

In this section we detail our proposal and modifications of Peikert’s protocol². For the same reasons as described in [20] we opt for an unauthenticated key-exchange protocol; the protection of stored transcripts against future decryption using quantum computers is much more urgent than post-quantum authentication. Authenticity will most likely be achievable in the foreseeable future using proven pre-quantum signatures and attacks on the signature will not compromise previous communication. Additionally, by not designing or instantiating a lattice-based authenticated key-exchange protocol (see [33, 85]) we reduce the complexity of the key-exchange protocol and simplify the choice of parameters. We actually see it as an advantage to decouple key exchange and authentication as it allows a protocol designer to choose the optimal algorithm for both tasks (e.g., an ideal-lattice-based key exchange and a hash-based signature like [16] for authentication). Moreover, this way the design, security level, and parameters of the key-exchange scheme are not constrained by requirements introduced by the authentication part.

Parameter choices. A high-level description of our proposal is given in Protocol 2 and as in [20, 77] all polynomials except for $\mathbf{r} \in \mathcal{R}_4$ are defined in the ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $n = 1024$ and $q = 12289$. We decided to keep the dimension $n = 1024$ as in [20] to be able to achieve appropriate long-term security. As polynomial arithmetic is fast and also scales better (doubling n roughly doubles the time required for a polynomial multiplication), our choice of n appears to be acceptable from a performance point of view. We chose the modulus $q = 12289$ as it is the smallest prime for which it holds that $q \equiv 1 \pmod{2n}$ so that the number-theoretic transform (NTT) can be realized efficiently and that we can transfer polynomials in NTT encoding (see Section 7).

²For the TLS use-case and for compatibility with BNCS [20] the key exchange is initiated by the server. However, in different scenarios the roles of the server and client can be exchanged.

As the security level grows with the noise-to-modulus ratio, it makes sense to choose the modulus as small as possible, improving compactness and efficiency together with security. The choice is also appealing as the prime is already used by some implementations of Ring-LWE encryption [29, 63, 81] and BLISS signatures [31, 78]; thus sharing of some code (or hardware modules) between our proposal and an implementation of BLISS would be possible.

Noise distribution and reconciliation. Notably, we also change the distribution of the LWE secret and error and replace discrete Gaussians by the centered binomial distribution ψ_k of parameter $k = 16$ (see Section 4). The reason is that it turned out to be challenging to implement a discrete Gaussian sampler efficiently *and* protected against timing attacks (see [20] and Section 5). On the other hand, sampling from the centered binomial distribution is easy and does not require high-precision computations or large tables as one may sample from ψ_k by computing $\sum_{i=0}^k b_i - b'_i$, where the $b_i, b'_i \in \{0, 1\}$ are uniform independent bits. The distribution ψ_k is centered (its mean is 0), has variance $k/2$ and for $k = 16$ this gives a standard deviation of $\xi = \sqrt{16/2}$. Contrary to [20, 77] we hash the output of the reconciliation mechanism, which makes a distinguishing attack irrelevant and allows us to argue security for the modified error distribution.

Moreover, we generalize Peikert’s reconciliation mechanism using an analog error-correction approach (see Section 5). The design rationale is that we only want to transmit a 256-bit key but have $n = 1024$ coefficients to encode data into. Thus we encode one key bit into four coefficients; by doing so we achieve increased error resilience which in turn allows us to use larger noise for better security.

Short-term public parameters. NEWHOPE does not rely on a globally chosen public parameter \mathbf{a} as the efficiency increase in doing so is not worth the measures that have to be taken to allow trusted generation of this value and the defense against backdoors [13]. Moreover, this approach avoids the rather uncomfortable situation that all connections rely on a single instance of a lattice problem (see Section 3) in the flavor of the “Logjam” DLP attack [1].

No key caching. For ephemeral Diffie-Hellman key-exchange in TLS it is common for servers to cache a key pair for a short time to increase performance. For example, according to [24], Microsoft’s SChannel library caches ephemeral keys for 2 hours. We remark that for the lattice-based key exchange described in [77], for the key exchange described in [20], and also for the key exchange described in this paper, such short-term caching would be disastrous for security. Indeed, it is crucial that

both parties use fresh secrets for each instantiation (thus the performance of the noise sampling is crucial). As short-term key caching typically happens on higher layers of TLS libraries than the key-exchange implementation itself, we stress that particular care needs to be taken to eliminate such caching when switching from ephemeral (elliptic-curve) Diffie-Hellman key exchange to post-quantum lattice-based key exchange. This issue is discussed in more detail in [32].

One could enable key caching with a transformation from the CPA-secure key exchange to a CCA-secure key exchange as outlined by Peikert in [77, Section 5]. Note that such a transform would furthermore require changes to the noise distribution to obtain a failure probability that is negligible in the cryptographic sense.

3 Preventing backdoors and all-for-the-price-of-one attacks

One serious concern about the original design [20] is the presence of the polynomial \mathbf{a} as a fixed system parameter. As described in Protocol 2, our proposal includes pseudorandom generation of this parameter for every key exchange. In the following we discuss the reasons for this decision.

Backdoor. In the worst scenario, the fixed parameter \mathbf{a} could be backdoored. For example, inspired by NTRU trapdoors [50, 83], a dishonest authority may choose mildly small \mathbf{f}, \mathbf{g} such that $\mathbf{f} = \mathbf{g} = 1 \bmod p$ for some prime $p \geq 4 \cdot 16 + 1$ and set $\mathbf{a} = \mathbf{g}\mathbf{f}^{-1} \bmod q$. Then, given $(\mathbf{a}, \mathbf{b} = \mathbf{a}\mathbf{s} + \mathbf{e})$, the attacker can compute $\mathbf{b}\mathbf{f} = \mathbf{a}\mathbf{f}\mathbf{s} + \mathbf{f}\mathbf{e} = \mathbf{g}\mathbf{s} + \mathbf{f}\mathbf{e} \bmod q$, and, because $\mathbf{g}, \mathbf{s}, \mathbf{f}, \mathbf{e}$ are small enough, compute $\mathbf{g}\mathbf{s} + \mathbf{f}\mathbf{e}$ in \mathbb{Z} . From this he can compute $\mathbf{t} = \mathbf{s} + \mathbf{e} \bmod p$ and, because the coefficients of \mathbf{s} and \mathbf{e} are smaller than 16, their sums are in $[-2 \cdot 16, 2 \cdot 16]$: knowing them modulo $p \geq 4 \cdot 16 + 1$ is knowing them in \mathbb{Z} . It now only remains to compute $(\mathbf{b} - \mathbf{t}) \cdot (\mathbf{a} - 1)^{-1} = (\mathbf{a}\mathbf{s} - \mathbf{s}) \cdot (\mathbf{a} - 1)^{-1} = \mathbf{s} \bmod q$ to recover the secret \mathbf{s} .

One countermeasure against such backdoors is the “nothing-up-my-sleeve” process, which would, for example, choose \mathbf{a} as the output of a hash function on a common universal string like the digits of π . Yet, even this process may be partially abused [13], and when not strictly required it seems preferable to avoid it.

All-for-the-price-of-one attacks. Even if this common parameter has been honestly generated, it is still rather uncomfortable to have the security of all connections rely on a single instance of a lattice problem. The scenario is an entity that discovers an unforeseen cryptanalytic algorithm, making the required lattice reduction still very costly, but say, not impossible in a year of computation, given its outstanding computational power. By finding *once* a good enough basis of the lattice $\Lambda =$

Parameters: $q = 12289 < 2^{14}$, $n = 1024$	
Error distribution: ψ_{16}	
Alice (server)	Bob (client)
$seed \xleftarrow{\$} \{0, 1\}^{256}$	
$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$	
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_{16}^n$	$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$
$\mathbf{b} \leftarrow \mathbf{a}\mathbf{s} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, seed)}$
$\mathbf{v}' \leftarrow \mathbf{u}\mathbf{s}$	$\xleftarrow{(\mathbf{u}, \mathbf{r})}$
$v \leftarrow \text{Rec}(\mathbf{v}', \mathbf{r})$	$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$
$\mu \leftarrow \text{SHA3-256}(v)$	$\mathbf{u} \leftarrow \mathbf{a}\mathbf{s}' + \mathbf{e}'$
	$\mathbf{v} \leftarrow \mathbf{b}\mathbf{s}' + \mathbf{e}''$
	$\mathbf{r} \xleftarrow{\$} \text{HelpRec}(\mathbf{v})$
	$v \leftarrow \text{Rec}(\mathbf{v}, \mathbf{r})$
	$\mu \leftarrow \text{SHA3-256}(v)$

Protocol 2: Our Scheme. For the definitions of HelpRec and Rec see Section 5. For the definition of encodings and the definition of Parse see Section 7.

$\{(a, 1)x + (q, 0)y \mid x, y \in \mathcal{R}\}$, this entity could then compromise *all* communications, using for example Babai’s decoding algorithm [7].

This idea of massive precomputation that is only dependent on a fixed parameter \mathbf{a} and then afterwards can be used to break all key exchanges is similar in flavor to the 512-bit “Logjam” DLP attack [1]. This attack was only possible in the required time limit because most TLS implementations use fixed primes for Diffie-Hellman. One of the recommended mitigations by the authors of [1] is to avoid fixed primes.

Against all authority. Fortunately, all those pitfalls can be avoided by having the communicating parties generate a fresh \mathbf{a} at each instance of the protocol (as we propose). If in practice it turns out to be too expensive to generate \mathbf{a} for every connection, it is also possible to cache \mathbf{a} on the server side³ for, say a few hours without significantly weakening the protection against all-for-the-price-of-one attacks. Additionally, the performance impact of generating \mathbf{a} is reduced by sampling \mathbf{a} uniformly directly in NTT format (recalling that the NTT is a one-to-one map), and by transferring only a short 256-bit seed for \mathbf{a} (see Section 7).

A subtle question is to choose an appropriate primitive to generate a “random-looking” polynomial \mathbf{a} out of a short seed. For a security reduction, it seems to the authors that there is no way around the (non-programmable) random oracle model (ROM). It is argued in [34] that such a requirement is in practice an overkill, and that any pseudorandom generator (PRG) should also work. And while it is an interesting question how such a reasonable pseudo-random generator would interact with our lattice assumption, the cryptographic

notion of a PRG is *not* helpful to argue security. Indeed, it is an easy exercise⁴ to build (under the NTRU assumption) a “backdoored” PRG that is, formally, a legitimate PRG, but that makes our scheme insecure.

Instead, we prefer to base ourselves on a standard cryptographic hash-function, which is the typical choice of an “instantiation” of the ROM. As a suitable option we see Keccak [19], which has recently been standardized as SHA3 in FIPS-202 [72], and which offers extendable-output functions (XOF) named SHAKE. This avoids costly external iteration of a regular hash function and directly fits our needs.

We use SHAKE-128 for the generation of \mathbf{a} , which offers 128-bits of (post-quantum) security against collisions and preimage attacks. With only a small performance penalty we could have also chosen SHAKE-256, but we do not see any reason for such a choice, in particular because neither collisions nor preimages lead to an attack against the proposed scheme.

4 Choice of the error distribution

On non-Gaussian errors. In works like [20, 29, 81], a significant algorithmic effort is devoted to sample from a discrete Gaussian distribution to a rather high precision. In the following we argue that such effort is not necessary and motivate our choice of a centered binomial ψ_k as error distribution.

Indeed, we recall that the original worst-case to average-case reductions for LWE [80] and Ring-

³But recall that the secrets $\mathbf{s}, \mathbf{e}, \mathbf{s}', \mathbf{s}', \mathbf{e}''$ have to be sampled fresh for every connection.

⁴Consider a secure PRG p , and parse its output $p(\text{seed})$ as two small polynomial (\mathbf{f}, \mathbf{g}) : an NTRU secret-key. Define $p'(\text{seed}) = \mathbf{g}\mathbf{f}^{-1} \bmod q$: under the decisonal NTRU assumption, p' is still a secure PRG. Yet revealing the seed does reveal (\mathbf{f}, \mathbf{g}) and provides a backdoor as detailed above.

LWE [67] state hardness for *continuous Gaussian* distributions (and therefore also trivially apply to *rounded Gaussian*, which differ from discrete Gaussians). This also extends to discrete Gaussians [21] but such proofs are not necessarily intended for direct implementations. We recall that the use of discrete Gaussians (or other distributions with very high-precision sampling) is only crucial for signatures [65] and lattice trapdoors [39], to provide zero-knowledgeness.

The following Theorem states that choosing ψ_k as error distribution in Protocol 2 does not significantly decrease security compared to a rounded Gaussian distribution with the same standard deviation $\sigma = \sqrt{16/2}$.

Theorem 4.1 *Let ξ be the rounded Gaussian distribution of parameter $\sigma = \sqrt{8}$, that is, the distribution of $\lfloor \sqrt{8} \cdot x \rfloor$ where x follows the standard normal distribution. Let \mathcal{P} be the idealized version of Protocol 2, where the distribution ψ_{16} is replaced by ξ . If an (unbounded) algorithm, given as input the transcript of an instance of Protocol 2 succeeds in recovering the pre-hash key v with probability p , then it would also succeed against \mathcal{P} with probability at least*

$$q \geq p^{9/8}/26.$$

Proof See Appendix B in the full version of this paper.

As explained in Section 6, our choice of parameters leaves a comfortable margin to the targeted 128 bits of post-quantum security, which accommodates for the slight loss in security indicated by Theorem 4.1. Even more important from a practical point of view is that no known attack makes use of the difference in error distribution; what matters for attacks are entropy and standard deviation.

Simple implementation. We remark that sampling from the centered binomial distribution ψ_{16} is rather trivial in hardware and software, given the availability of a uniform binary source. Additionally, the implementation of this sampling algorithm is much easier to protect against timing attacks as no large tables or data-dependent branches are required (cf. to the issues caused by the table-based approach used in [20]).

5 Improved error-recovery mechanism

In most of the literature, Ring-LWE encryption allows to encrypt one bit per coordinate of the ciphertext. It is also well known how to encrypt multiple bits per coordinate by using a larger modulus-to-error ratio (and therefore decreasing the security for a fixed dimension n). However, in the context of exchanging a symmetric key (of, say, 256 bits), we end up having a message space larger

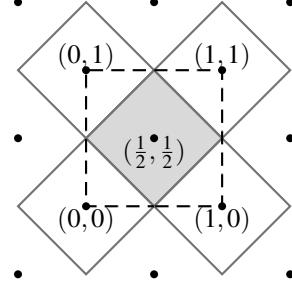


Figure 1: The lattice \tilde{D}_2 with Voronoi cells

than necessary and thus want to encrypt *one bit in multiple coordinates*.

In [79] Pöppelmann and Güneysu introduced a technique to encode one bit into two coordinates, and verified experimentally that it led to a better error tolerance. This allows to either increase the error and therefore improve the security of the resulting scheme or to decrease the probability of decryption failures. In this section we propose a generalization of this technique in dimension 4. We start with an intuitive description of the approach in 2 dimensions and then explain what changes in 4 dimensions. Appendices C and D in the full version of this paper give a thorough mathematical description together with a rigorous analysis.

Let us first assume that both client and server have the same vector $\mathbf{x} \in [0, 1)^2 \subset \mathbb{R}^2$ and want to map this vector to a single bit. Mapping polynomial coefficients from $\{0, \dots, q-1\}$ to $[0, 1)$ is easily accomplished through a division by q .

Now consider the lattice \tilde{D}_2 with basis $\{(0, 1), (\frac{1}{2}, \frac{1}{2})\}$. This lattice is a scaled version of the root lattice D_2 , specifically, $\tilde{D}_2 = \frac{1}{2} \cdot D_2$. Part of \tilde{D}_2 is depicted in Figure 1; lattice points are shown together with their Voronoi cells and the possible range of the vector \mathbf{x} is marked with dashed lines. Mapping \mathbf{x} to one bit is done by finding the closest-vector $v \in \tilde{D}_2$. If $v = (\frac{1}{2}, \frac{1}{2})$ (i.e., \mathbf{x} is in the grey Voronoi cell), then the output bit is 1; if $v \in \{(0, 0), (0, 1), (1, 0), (1, 1)\}$ (i.e., \mathbf{x} is in a white Voronoi cell) then the output bit is 0.

This map may seem like a fairly complex way to map from a vector to a bit. However, recall that client and server only have a noisy version of \mathbf{x} , i.e., the client has a vector \mathbf{x}_c and the server has a vector \mathbf{x}_s . Those two vectors are close, but they are not the same and can be on different sides of a Voronoi cell border.

Error reconciliation. The approach described above now allows for an efficient solution to solve this agreement-from-noisy-data problem. The idea is that one of the two participants (in our case the client) sends as a *reconciliation vector* the difference of his vector \mathbf{x}_c

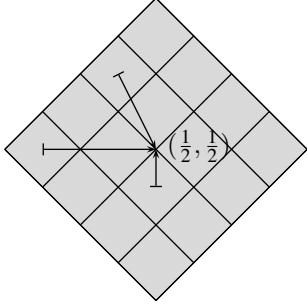


Figure 2: Splitting of the Voronoi cell of $(\frac{1}{2}, \frac{1}{2})$ into $2^{rd} = 16$ sub-cells, some with their corresponding difference vector to the center

and the center of its Voronoi cell (i.e., the point in the lattice). The server adds this difference vector to \mathbf{x}_s and thus moves away from the border towards the center of the correct Voronoi cell. Note that an eavesdropper does not learn anything from the reconciliation information: the client tells the difference to a lattice point, but not whether this is a lattice point producing a zero bit or a one bit.

This approach would require sending a full additional vector; we can reduce the amount of reconciliation information through r -bit discretization. The idea is to split each Voronoi cell into 2^{dr} sub-cells and only send in which of those sub-cells the vector x_c is. Both participants then add the difference of the center of the sub-cell and the lattice point. This is illustrated for $r = 2$ and $d = 2$ in Figure 2.

Blurring the edges. Figure 1 may suggest that the probability of \mathbf{x} being in a white Voronoi cell is the same as for \mathbf{x} being in the grey Voronoi cell. This would be the case if \mathbf{x} actually followed a continuous uniform distribution. However, the coefficients of \mathbf{x} are discrete values in $\{0, \frac{1}{q}, \dots, \frac{q-1}{q}\}$ and with the protocol described so far, the bits of \mathbf{v} would have a small bias. The solution is to add, with probability $\frac{1}{2}$, the vector $(\frac{1}{2q}, \frac{1}{2q})$ to \mathbf{x} before running the error reconciliation. This has close to no effect for most values of \mathbf{x} , but, with probability $\frac{1}{2}$ moves \mathbf{x} to another Voronoi cell if it is very close to one side of a border. Appendix E in the full version of this paper gives a graphical intuition for this trick in two dimensions and with $q = 9$. The proof that it indeed removes all biases in the key is given in Lemma C.2. in the full version of this paper.

From 2 to 4 dimensions. When moving from the 2-dimensional case considered above to the 4-dimensional case used in our protocol, not very much needs to change. The lattice \tilde{D}_2 becomes the lattice \tilde{D}_4 with basis $\mathbf{B} = (\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, \mathbf{g})$, where \mathbf{u}_i are the canonical basis vectors of

\mathbb{Z}^4 and $\mathbf{g}' = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$. The lattice \tilde{D}_4 is a rotated and scaled version of the root lattice D_4 . The Voronoi cells of this lattice are no longer 2-dimensional ‘‘diamonds’’, but 4-dimensional objects called icositetrachoron or 24-cells [61]. Determining in which cell a target point lies in is done using the closest vector algorithm $\text{CVP}_{\tilde{D}_4}$, and a simplified version of it, which we call `Decode`, gives the result modulo \mathbb{Z}^4 .

As in the 2-dimensional illustration in Figure 2, we are using 2-bit discretization; we are thus sending $r \cdot d = 8$ bits of reconciliation information per key bit.

Putting all of this together, we obtain the `HelpRec` function to compute the r -bit reconciliation information as

$$\text{HelpRec}(\mathbf{x}; b) = \text{CVP}_{\tilde{D}_4} \left(\frac{2^r}{q} (\mathbf{x} + b\mathbf{g}') \right) \bmod 2^r,$$

where $b \in \{0, 1\}$ is a uniformly chosen random bit. The corresponding function $\text{Rec}(\mathbf{x}, \mathbf{r}) = \text{Decode}(\frac{1}{q}\mathbf{x} - \frac{1}{2^r}\mathbf{Br})$ computes one key bit from a vector \mathbf{x} with 4 coefficients in \mathbb{Z}_q and a reconciliation vector $\mathbf{r} \in \{0, 1, 2, 3\}^4$. The algorithms $\text{CVP}_{\tilde{D}_4}$ and `Decode` are listed as Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 $\text{CVP}_{\tilde{D}_4}(\mathbf{x} \in \mathbb{R}^4)$

Ensure: An integer vector \mathbf{z} such that \mathbf{Bz} is a closest vector to \mathbf{x} : $\mathbf{x} - \mathbf{Bz} \in \mathcal{V}$

- 1: $\mathbf{v}_0 \leftarrow \lfloor \mathbf{x} \rfloor$
 - 2: $\mathbf{v}_1 \leftarrow \lfloor \mathbf{x} - \mathbf{g}' \rfloor$
 - 3: $k \leftarrow (\|\mathbf{x} - \mathbf{v}_0\|_1 < 1) ? 0 : 1$
 - 4: $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)^t \leftarrow \mathbf{v}_k$
 - 5: **return** $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, k)^t + v_3 \cdot (-1, -1, -1, 2)^t$
-

Algorithm 2 $\text{Decode}(\mathbf{x} \in \mathbb{R}^4 / \mathbb{Z}^4)$

Ensure: A bit k such that $k\mathbf{g}'$ is a closest vector to $\mathbf{x} + \mathbb{Z}^4$: $\mathbf{x} - k\mathbf{g}' \in \mathcal{V} + \mathbb{Z}^4$

- 1: $\mathbf{v} = \mathbf{x} - \lfloor \mathbf{x} \rfloor$
 - 2: **return** 0 if $\|\mathbf{v}\|_1 \leq 1$ and 1 otherwise
-

Finally it remains to remark that even with this reconciliation mechanism client and server do not always agree on the same key. Lemma D in the full version of this paper provides a detailed analysis of the failure probability of the key agreement and shows that it is smaller than 2^{-60} .

6 Post-quantum security analysis

In [20] the authors chose Ring-LWE for a ring of rank $n = 1024$, while most previous instantiations of the Ring-LWE encryption scheme, like the ones in [29, 42, 63, 79],

chose substantially smaller rank $n = 256$ or $n = 512$. It is argued that it is unclear if dimension 512 can offer post-quantum security. Yet, the concrete post-quantum security of LWE-based schemes has not been thoroughly studied, as far as we know. In this section we propose such a (very pessimistic) concrete analysis. In particular, our analysis reminds us that the security depends as much on q and its ratio with the error standard deviation ζ as it does on the dimension n . That means that our effort of optimizing the error recovery and its analysis not only improves efficiency but also offers superior security.

Security level over-shoot? With all our improvements, it would be possible to build a scheme with $n = 512$ (and $k = 24$, $q = 12289$) and to obtain security somewhat similar to the one of [20, 42], and therefore further improve efficiency. We call this variant JARJAR and details are provided in Appendix A of the full version of this paper. Nevertheless, as history showed us with RSA-512 [28], the standardization and deployment of a scheme awakens further cryptanalytic effort. In particular, NEWHOPE could withstand a dimension-halving attack in the line of [36, Sec 8.8.1] based on the Gentry-Szydlo algorithm [40, 60] or the subfield approach of [2]. Note that so far, such attacks are only known for principal ideal lattices or NTRU lattices, and there are serious obstructions to extend them to Ring-LWE, but such precaution seems reasonable until lattice cryptanalysis stabilizes.

We provide the security and performance analysis of JARJAR in Appendix A of the full version of this paper mostly for comparison with other lower-security proposals. We strongly recommend NEWHOPE for any immediate applications, and advise against using JARJAR until concrete cryptanalysis of lattice-based cryptography is better understood.

6.1 Methodology: the core SVP hardness

We analyze the hardness of Ring-LWE as an LWE problem, since, so far, the best known attacks do not make use of the ring structure. There are many algorithms to consider in general (see the survey [3]), yet many of those are irrelevant for our parameter set. In particular, because there are only $m = n$ samples available one may rule out BKW types of attacks [53] and linearization attacks [6]. This essentially leaves us with two BKZ [26, 82] attacks, usually referred to as primal and dual attacks that we will briefly recall below.

The algorithm BKZ proceeds by reducing a lattice basis using an SVP oracle in a smaller dimension b . It is known [47] that the number of calls to that oracle remains polynomial, yet concretely evaluating the number of calls is rather painful, and this is subject to new heuristic ideas [25, 26]. We choose to ignore this polynomial

factor, and rather evaluate only the *core SVP hardness*, that is the cost of *one call* to an SVP oracle in dimension b , which is clearly a pessimistic estimation (from the defender's point of view).

6.2 Enumeration versus quantum sieve

Typical implementations [23, 26, 35] use an enumeration algorithm as this SVP oracle, yet this algorithm runs in super-exponential time. On the other hand, the sieve algorithms are known to run in exponential time, but are so far slower in practice for accessible dimensions $b \approx 130$. We choose the latter to predict the core hardness and will argue that for the targeted dimension, enumerations are expected to be greatly slower than sieving.

Quantum sieve. A lot of recent work has pushed the efficiency of the original lattice sieve algorithms [69, 75], improving the heuristic complexity from $(4/3)^{b+o(b)} \approx 2^{0.415b}$ down to $\sqrt{3/2}^{b+o(b)} \approx 2^{0.292b}$ (see [10, 55]). The hidden sub-exponential factor is known to be much greater than one in practice, so again, estimating the cost ignoring this factor leaves us with a significant pessimistic margin.

Most of those algorithms have been shown [54, 56] to benefit from Grover's quantum search algorithm, bringing the complexity down to $2^{0.265b}$. It is unclear if further improvements are to be expected, yet, because all those algorithms require classically building lists of size $\sqrt{4/3}^{b+o(b)} \approx 2^{0.2075b}$, it is very plausible that the best quantum SVP algorithm would run in time greater than $2^{0.2075b}$.

Irrelevance of enumeration for our analysis. In [26], predictions of the cost of solving SVP classically using the most sophisticated heuristic enumeration algorithms are given. For example, solving SVP in dimension 100 requires visiting about 2^{39} nodes, and 2^{134} nodes in dimension 250. Because this enumeration is a backtracking algorithm, it does benefit from the recent quasi-quadratic speedup [70], decreasing the quantum cost to about at least 2^{20} to 2^{67} operations as the dimension increases from 100 to 250.

On the other hand, our best-known attack bound $2^{0.265b}$ gives a cost of 2^{66} in dimension 250, and the best plausible attack bound $2^{0.2075b} \approx 2^{39}$. Because enumeration is super-exponential (both in theory and practice), its cost will be worse than our bounds in dimension larger than 250 and we may safely ignore this kind of algorithm.⁵

⁵The numbers are taken from the latest full version of [26] available at http://www.di.ens.fr/~ychen/research/Full_BKZ.pdf.

6.3 Primal attack

The primal attack consists of constructing a unique-SVP instance from the LWE problem and solving it using BKZ. We examine how large the block dimension b is required to be for BKZ to find the unique solution. Given the matrix LWE instance $(\mathbf{A}, \mathbf{b} = \mathbf{As} + \mathbf{e})$ one builds the lattice $\Lambda = \{\mathbf{x} \in \mathbb{Z}^{m+n+1} : (\mathbf{A} - \mathbf{I}_m) - \mathbf{b}\mathbf{x} = \mathbf{0} \bmod q\}$ of dimension $d = m+n+1$, volume q^m , and with a unique-SVP solution $\mathbf{v} = (\mathbf{s}, \mathbf{e}, 1)$ of norm $\lambda \approx \zeta\sqrt{n+m}$. Note that the number of used samples m may be chosen between 0 and $2n$ in our case and we numerically optimize this choice.

Success condition. We model the behavior of BKZ using the geometric series assumption (which is known to be optimistic from the attacker’s point of view), that finds a basis whose Gram-Schmidt norms are given by $\|\mathbf{b}_i^*\| = \delta^{d-2i-1} \cdot \text{Vol}(\Lambda)^{1/d}$ where $\delta = ((\pi b)^{1/b} \cdot b/2\pi e)^{1/2(b-1)}$ [3, 25]. The unique short vector \mathbf{v} will be detected if the projection of \mathbf{v} onto the vector space spanned by the last b Gram-Schmidt vectors is shorter than \mathbf{b}_{d-b}^* . Its projected norm is expected to be $\zeta\sqrt{b}$, that is the attack is successful if and only if

$$\zeta\sqrt{b} \leq \delta^{2b-d-1} \cdot q^{m/d}. \quad (1)$$

6.4 Dual attack

The dual attack consists of finding a short vector in the dual lattice $\mathbf{w} \in \Lambda' = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}^m \times \mathbb{Z}^n : \mathbf{A}^t \mathbf{x} = \mathbf{y} \bmod q\}$. Assume we have found a vector (\mathbf{x}, \mathbf{y}) of length ℓ and compute $z = \mathbf{v}^t \cdot \mathbf{b} = \mathbf{v}^t \mathbf{As} + \mathbf{v}^t \mathbf{e} = \mathbf{w}^t \mathbf{s} + \mathbf{v}^t \mathbf{e} \bmod q$ which is distributed as a Gaussian of standard deviation $\ell\zeta$ if (\mathbf{A}, \mathbf{b}) is indeed an LWE sample (otherwise it is uniform mod q). Those two distributions have maximal variation distance bounded by⁶ $\epsilon = 4 \exp(-2\pi^2\tau^2)$ where $\tau = \ell\zeta/q$, that is, given such a vector of length ℓ one has an advantage ϵ against decision-LWE.

The length ℓ of a vector given by the BKZ algorithm is given by $\ell = \|\mathbf{b}_0\|$. Knowing that Λ' has dimension $d = m+n$ and volume q^n we get $\ell = \delta^{d-1} q^{n/d}$. Therefore, obtaining an ϵ -distinguisher requires running BKZ with block dimension b where

$$-2\pi^2\tau^2 \geq \ln(\epsilon/4). \quad (2)$$

Note that small advantages ϵ are not relevant since the agreed key is hashed: an attacker needs an advantage of at least $1/2$ to significantly decrease the search space of the agreed key. He must therefore amplify his success

⁶A preliminary version of this paper contained a bogus formula for ϵ leading to under-estimating the cost of the dual attack. Correcting this formula leads to better security claim, and almost similar cost for the primal and dual attacks.

Attack	m	b	Known Classical	Known Quantum	Best Plausible
BCNS proposal [20]: $q = 2^{32} - 1$, $n = 1024$, $\zeta = 3.192$					
Primal	1062	296	86	78	61
Dual	1055	296	86	78	61
NTRUENCRYPT [49]: $q = 2^{12}$, $n = 743$, $\zeta \approx \sqrt{2/3}$					
Primal	613	603	176	159	125
Dual	635	600	175	159	124
JARJAR: $q = 12289$, $n = 512$, $\zeta = \sqrt{12}$					
Primal	623	449	131	119	93
Dual	602	448	131	118	92
NEWHOPE: $q = 12289$, $n = 1024$, $\zeta = \sqrt{8}$					
Primal	1100	967	282	256	200
Dual	1099	962	281	255	199

Table 1: Core hardness of NEWHOPE and JARJAR and selected other proposals from the literature. The value b denotes the block dimension of BKZ, and m the number of used samples. Cost is given in \log_2 and is the smallest cost for all possible choices of m and b . Note that our estimation is very optimistic about the abilities of the attacker so that our result for the parameter set from [20] *does not* indicate that it can be broken with $\approx 2^{80}$ bit operations, given today’s state-of-the-art in cryptanalysis.

probability by building about $1/\epsilon^2$ many such short vectors. Because the sieve algorithms provide $2^{0.2075b}$ vectors, the attack must be repeated at least R times where

$$R = \max(1, 1/(2^{0.2075b} \epsilon^2)).$$

This makes the conservative assumption that all the vectors provided by the Sieve algorithm are as short as the shortest one.

6.5 Security claims

According to our analysis, we claim that our proposed parameters offer at least (and quite likely with a large margin) a post-quantum security of 128 bits. The cost of the primal attack and dual attacks (estimated by our script `scripts/PQsecurity.py`) are given in Table 1. For comparison we also give a lower bound on the security of [20] and do notice a significantly improved security in our proposal. Yet, because of the numerous pessimistic assumption made in our analysis, we do not claim any quantum attacks reaching those bounds.

Most other RLWE proposals achieve considerably lower security than NEWHOPE; for example, the highest-security parameter set used for RLWE encryption in [42] is very similar to the parameters of JARJAR. The situation is different for NTRUENCRYPT, which has been instantiated with parameters that achieve about 128 bits of security according to our analysis⁷.

⁷For comparison we view the NTRU key-recovery as an homoge-

Specifically, we refer to NTRUENCRYPT with $n = 743$ as suggested in [49]. A possible advantage of NTRUENCRYPT compared to NEWHOPE is somewhat smaller message sizes, however, this advantage becomes very small when scaling parameters to achieve a similar security margin as NEWHOPE. The large downside of using NTRUENCRYPT for ephemeral key exchange is the cost for key generation. The implementation of NTRUENCRYPT with $n = 743$ in eBACS [17] takes about an order of magnitude longer for key generation alone than NEWHOPE takes in total. Also, unlike our NEWHOPE software, this NTRUENCRYPT software is not protected against timing attacks; adding such protection would presumably incur a significant overhead.

7 Implementation

In this section we provide details on the encodings of messages and describe our portable reference implementation written in C, as well as an optimized implementation targeting architectures with AVX vector instructions.

7.1 Encodings and generation of a

The key-exchange protocol described in Protocol 1 and also our protocol as described in Protocol 2 exchange messages that contain mathematical objects (in particular, polynomials in \mathcal{R}_q). *Implementations* of these protocols need to exchange messages in terms of byte arrays. As we will describe in the following, the choice of encodings of polynomials to byte arrays has a serious impact on performance. We use an encoding of messages that is particularly well-suited for implementations that make use of quasi-linear NTT-based polynomial multiplication.

Definition of NTT and NTT^{-1} . The NTT is a tool commonly used in implementations of ideal lattice-based cryptography [29, 42, 63, 79]. For some background on the NTT and the description of fast software implementations we refer to [46, 68]. In general, fast quasi-logarithmic algorithms exist for the computation of the NTT and a polynomial multiplication can be performed by computing $\mathbf{c} = \text{NTT}^{-1}(\text{NTT}(\mathbf{a}) \circ \text{NTT}(\mathbf{b}))$ for $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathcal{R}$. An NTT targeting ideal lattices defined in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ can be implemented very efficiently if n is a power of two and q is a prime for which it holds that $q \equiv 1 \pmod{2n}$. This way a primitive n -th root of unity ω and its square root γ exist. By multiplying coefficient-wise by powers of $\gamma = \sqrt{\omega} \pmod{q}$ before

neous Ring-LWE instance. We do not take into account the combinatorial vulnerabilities [51] induced by the fact that secrets are ternary. We note that NTRU is a potentially a weaker problem than Ring-LWE: it is in principle subject to a subfield-lattice attack [2], but the parameters proposed for NTRUENCRYPT are immune.

the NTT computation and after the reverse transformation by powers of γ^{-1} , no zero padding is required and an n -point NTT can be used to transform a polynomial with n coefficients.

For a polynomial $\mathbf{g} = \sum_{i=0}^{1023} g_i X^i \in \mathcal{R}_q$ we define

$$\begin{aligned} \text{NTT}(\mathbf{g}) &= \hat{\mathbf{g}} = \sum_{i=0}^{1023} \hat{g}_i X^i, \text{ with} \\ \hat{g}_i &= \sum_{j=0}^{1023} \gamma^j g_j \omega^{ij}, \end{aligned}$$

where we fix the n -th primitive root of unity to $\omega = 49$ and thus $\gamma = \sqrt{\omega} = 7$. Note that in our implementation we use an in-place NTT algorithm which requires bit-reversal operations. As an optimization, our implementations skips these bit-reversals for the forward transformation as all inputs are only random noise. This optimization is transparent to the protocol and for simplicity omitted in the description here.

The function NTT^{-1} is the inverse of the function NTT. The computation of NTT^{-1} is essentially the same as the computation of NTT, except that it uses $\omega^{-1} \pmod{q} = 1254$, multiplies by powers of $\gamma^{-1} \pmod{q} = 8778$ after the summation, and also multiplies each coefficient by the scalar $n^{-1} \pmod{q} = 12277$ so that

$$\begin{aligned} \text{NTT}^{-1}(\hat{\mathbf{g}}) &= \mathbf{g} = \sum_{i=0}^{1023} g_i X^i, \text{ with} \\ g_i &= n^{-1} \gamma^{-i} \sum_{j=0}^{1023} \hat{g}_j \omega^{-ij}. \end{aligned}$$

The inputs to NTT^{-1} are *not* just random noise, so inside NTT^{-1} our software has to perform the initial bit reversal, making NTT^{-1} slightly more costly than NTT.

Definition of Parse. The public parameter \mathbf{a} is generated from a 256-bit seed through the extendable-output function SHAKE-128 [72, Sec. 6.2]. The output of SHAKE-128 is considered as an array of 16-bit, unsigned, little-endian integers. Each of those integers is used as a coefficient of \mathbf{a} if it is smaller than $5q$ and rejected otherwise. The first such 16-bit integer is used as the coefficient of X^0 , the next one as coefficient of X^1 and so on. Earlier versions of this paper described a slightly different way of rejection sampling for coefficients of \mathbf{a} . The more efficient approach adopted in this final version was suggested independently by Gueron and Schlieker in [45] and by Yawning Angel in [5]. However, note that a reduction modulo q of the coefficients of \mathbf{a} as described in [45] and [5] is not necessary; both our implementations can handle coefficients of \mathbf{a} in $\{0, \dots, 5q - 1\}$.

Due to a small probability of rejections, the amount of output required from SHAKE-128 depends on the seed –

what is required is $n = 1024$ coefficients that are smaller than $5q$. The minimal amount of output is thus 2 KB; the average amount is ≈ 2184.5 bytes. The resulting polynomial \mathbf{a} (denoted as $\hat{\mathbf{a}}$) is considered to be in NTT domain. This is possible because the NTT transforms uniform noise to uniform noise.

Using a variable amount of output from SHAKE-128 leaks information about \mathbf{a} through timing information. This is not a problem for most applications, since \mathbf{a} is public. As pointed out by Burdges in [22], such a timing leak of public information can be a problem when deploying NEWHOPE in anonymity networks like Tor. Appendix F in the full version of this paper describes an alternative approach for Parse, which is slightly more complex and slightly slower, but does not leak any timing information about \mathbf{a} .

The message format of $(\mathbf{b}, \text{seed})$ and (\mathbf{u}, \mathbf{r}) . With the definition of the NTT, we can now define the format of the exchanged messages. In both $(\mathbf{b}, \text{seed})$ and (\mathbf{u}, \mathbf{r}) the polynomial is transmitted in the NTT domain (as in works like [79, 81]). Polynomials are encoded as an array of 1792 bytes, in a compressed little-endian format. The encoding of seed is straight-forward as an array of 32 bytes, which is simply concatenated with the encoding of \mathbf{b} . Also the encoding of \mathbf{r} is fairly straight-forward: it packs four 2-bit coefficients into one byte for a total of 256 bytes, which are again simply concatenated with the encoding of \mathbf{u} . We denote these encodings to byte arrays as `encodeA` and `encodeB` and their inverses as `decodeA` and `decodeB`. For a description of our key-exchange protocol including encodings and with explicit NTT and NTT^{-1} transformations, see Protocol 3.

7.2 Portable C implementation

This paper is accompanied by a C reference implementation described in this section and an optimized implementation for Intel and AMD CPUs described in the next section. The main emphasis in the C reference implementation is on simplicity and portability. It does not use any floating-point arithmetic and outside the Keccak (SHA3-256 and SHAKE-128) implementation only needs 16-bit and 32-bit integer arithmetic. In particular, the error-recovery mechanism described in Section 5 is implemented with fixed-point (i.e., integer-) arithmetic. Furthermore, the C reference implementation does not make use of the division operator (/) and the modulo operator (%). The focus on simplicity and portability does not mean that the implementation is not optimized at all. On the contrary, we use it to illustrate various optimization techniques that are helpful to speed up the key exchange and are also of independent interest for implementers of other ideal-lattice-based schemes.

NTT optimizations. All polynomial coefficients are represented as unsigned 16-bit integers. Our in-place NTT implementation transforms from bit-reversed to natural order using Gentleman-Sande butterfly operations [27, 37]. One would usually expect that each NTT is preceded by a bit-reversal, but all inputs to NTT are noise polynomials that we can simply consider as being already bit-reversed; as explained earlier, the NTT^{-1} operation still involves a bit-reversal. The core of the NTT and NTT^{-1} operation consists of 10 layers of transformations, each consisting of 512 butterfly operations of the form described in Listing 2.

Montgomery arithmetic and lazy reductions. The performance of operations on polynomials is largely determined by the performance of NTT and NTT^{-1} . The main computational bottleneck of those operations are 5120 butterfly operations, each consisting of one addition, one subtraction and one multiplication by a precomputed constant. Those operations are in \mathbb{Z}_q ; recall that q is a 14-bit prime. To speed up the modular-arithmetic operations, we store all precomputed constants in Montgomery representation [71] with $R = 2^{18}$, i.e., instead of storing ω^i , we store $2^{18}\omega^i \pmod{q}$. After a multiplication of a coefficient g by some constant $2^{18}\omega^i$, we can then reduce the result r to $g\omega^i \pmod{q}$ with the fast Montgomery reduction approach. In fact, we do not always fully reduce modulo q , it is sufficient if the result of the reduction has at most 14 bits. The fast Montgomery reduction routine given in Listing 1a computes such a reduction to a 14-bit integer for any unsigned 32-bit integer in $\{0, \dots, 2^{32} - q(R-1) - 1\}$. Note that the specific implementation does not work for *any* 32-bit integer; for example, for the input $2^{32} - q(R-1) = 1073491969$ the addition $a=a+u$ causes an overflow and the function returns 0 instead of the correct result 4095. In the following we establish that this is not a problem for our software.

Aside from reductions after multiplication, we also need modular reductions after addition. For this task we use the “short Barrett reduction” [9] detailed in Listing 1b. Again, this routine does not fully reduce modulo q , but reduces any 16-bit unsigned integer to an integer of at most 14 bits which is congruent modulo q .

In the context of the NTT and NTT^{-1} , we make sure that inputs have coefficients of at most 14 bits. This allows us to avoid Barrett reductions after addition on every second level, because coefficients grow by at most one bit per level and the short Barrett reduction can handle 16-bit inputs. Let us turn our focus to the input of the Montgomery reduction (see Listing 2). Before subtracting $a[j+d]$ from t we need to add a multiple of q to avoid unsigned underflow. Coefficients never grow larger than 15 bits and $3 \cdot q = 36867 > 2^{15}$, so adding $3 \cdot q$ is sufficient. An upper bound on the

Parameters: $q = 12289 < 2^{14}$, $n = 1024$ Error distribution: ψ_{16}^n	
Alice (server)	Bob (client)
$seed \xleftarrow{\$} \{0, \dots, 255\}^{32}$ $\hat{\mathbf{a}} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$ $\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_{16}^n$ $\hat{\mathbf{s}} \leftarrow \text{NTT}(\mathbf{s})$ $\hat{\mathbf{b}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{s}} + \text{NTT}(\mathbf{e})$	$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_{16}^n$ $(\hat{\mathbf{b}}, seed) \leftarrow \text{decodeA}(m_a)$ $\hat{\mathbf{a}} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$ $\hat{\mathbf{t}} \leftarrow \text{NTT}(\mathbf{s}')$ $\hat{\mathbf{u}} \leftarrow \hat{\mathbf{a}} \circ \hat{\mathbf{t}} + \text{NTT}(\mathbf{e}')$ $\mathbf{v} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{b}} \circ \hat{\mathbf{t}}) + \mathbf{e}''$ $\mathbf{r} \xleftarrow{\$} \text{HelpRec}(\mathbf{v})$ $v \leftarrow \text{Rec}(\mathbf{v}, \mathbf{r})$ $\mu \leftarrow \text{SHA3-256}(v)$
	$m_a = \text{encodeA}(seed, \hat{\mathbf{b}})$ 1824 Bytes
$(\hat{\mathbf{u}}, \mathbf{r}) \leftarrow \text{decodeB}(m_b)$ $\mathbf{v}' \leftarrow \text{NTT}^{-1}(\hat{\mathbf{u}} \circ \hat{\mathbf{s}})$ $v \leftarrow \text{Rec}(\mathbf{v}', \mathbf{r})$ $\mu \leftarrow \text{SHA3-256}(v)$	$m_b = \text{encodeB}(\hat{\mathbf{u}}, \mathbf{r})$ 2048 Bytes

Protocol 3: Our proposed protocol including NTT and NTT^{-1} computations and sizes of exchanged messages; \circ denotes pointwise multiplication; elements in NTT domain are denoted with a hat (^)

expression $((\text{uint32_t})t + 3 * 12289 - a[j+d])$ is obtained if t is $2^{15} - 1$ and $a[j+d]$ is zero; we thus obtain $2^{15} + 3 \cdot q = 69634$. All precomputed constants are in $\{0, \dots, q - 1\}$, so the expression $(W * ((\text{uint32_t})t + 3 * 12289 - a[j+d]))$, the input to the Montgomery reduction, is at most $69634 \cdot (q - 1) = 855662592$ and thus safely below the maximum input that the Montgomery reduction can handle.

Listing 1 Reduction routines used in the reference implementation.

(a) Montgomery reduction ($R = 2^{18}$).

```
uint16_t mred(uint32_t a) {
    uint32_t u;
    u = (a * 12287);
    u &= ((1 << 18) - 1);
    a += u * 12289;
    return a >> 18;
}
```

(b) Short Barrett reduction.

```
uint16_t bred(uint16_t a) {
    uint32_t u;
    u = ((uint32_t)a * 5) >> 16;
    a -= u * 12289;
    return a;
}
```

Fast random sampling. As a first step before performing any operations on polynomials, both Alice and Bob need to expand the seed to the polynomial \mathbf{a} using SHAKE-128. The implementation we use is based on the “simple” implementation by Van Keer for the Kec-

Listing 2 The Gentleman-Sande butterfly inside odd levels of our NTT computation. All $a[j]$ and W are of type `uint16_t`.

```
W = omega[jTwiddle++];
t = a[j];
a[j] = bred(t + a[j+d]);
a[j+d] = mred(W * ((uint32_t)t + 3*12289 - a[j+d]));
```

cak permutation and slightly modified code taken from the “TweetFIPS202” implementation [18] for everything else.

The sampling of centered binomial noise polynomials is based on a fast PRG with a random seed from `/dev/urandom` followed by a quick summation of 16-bit chunks of the PRG output. Note that the choice of the PRG is a purely local choice that every user can pick independently based on the target hardware architecture and based on routines that are available anyway (for example, for symmetric encryption following the key exchange). Our C reference implementation uses ChaCha20 [12], which is fast, trivially protected against timing attacks, and is already in use by many TLS clients and servers [57, 58].

7.3 Optimized AVX2 implementation

Intel processors since the “Sandy Bridge” generation support Advanced Vector Extensions (AVX) that operate on vectors of 8 single-precision or 4 double-precision

floating-point values in parallel. With the introduction of the “Haswell” generation of CPUs, this support was extended also to 256-bit vectors of integers of various sizes (AVX2). It is not surprising that the enormous computational power of these vector instructions has been used before to implement very high-speed crypto (see, for example, [14, 16, 43]) and also our optimized reference implementation targeting Intel Haswell processors uses those instructions to speed up multiple components of the key exchange.

NTT optimizations. The AVX instruction set has been used before to speed up the computation of lattice-based cryptography, and in particular the number-theoretic transform. Most notably, Güneysu, Oder, Pöppelmann and Schwabe achieve a performance of only 4480 cycles for a dimension-512 NTT on Intel Sandy Bridge [46]. For arithmetic modulo a 23-bit prime, they represent coefficients as double-precision integers.

We experimented with multiple different approaches to speed up the NTT in AVX. For example, we vectorized the Montgomery arithmetic approach of our C reference implementation and also adapted it to a 32-bit-signed-integer approach. In the end it turned out that floating-point arithmetic beats all of those more sophisticated approaches, so we are now using an approach that is very similar to the approach in [46]. One computation of a dimension-1024 NTT takes 8448 cycles, unlike the numbers in [46] this does include multiplication by the powers of γ and unlike the numbers in [46], this excludes a bit-reversal.

Fast sampling. Intel Haswell processors support the AES-NI instruction set and for the local choice of noise sampling it is obvious to use those. More specifically, we use the public-domain implementation of AES-256 in counter mode written by Dolbeau, which is included in the SUPERCOP benchmarking framework [17]. Transformation from uniform noise to the centered binomial is optimized in AVX2 vector instructions operating on vectors of bytes and 16-bit integers.

For the computation of SHAKE-128 we use the same code as in the C reference implementation. One might expect that architecture-specific optimizations (for example, using AVX instructions) are able to offer significant speedups, but the benchmarks of the eBACS project [17] indicate that on Intel Haswell, the fastest implementation is the “simple” implementation by Van Keer that our C reference implementation is based on. The reasons that vector instructions are not very helpful for speeding up SHAKE (or, more generally, Keccak) are the inherently sequential nature and the 5×5 dimension of the state matrix that makes internal vectorization hard.

Error recovery. The 32-bit integer arithmetic used by the C reference implementation for HelpRec and Rec

is trivially 8-way parallelized with AVX2 instructions. With this vectorization, the cost for HelpRec is only 3404 cycles, the cost for Rec is only 2804 cycles.

8 Benchmarks and comparison

In the following we present benchmark results of our software. All benchmark results reported in Table 2 were obtained on an Intel Core i7-4770K (Haswell) running at 3491.953 MHz with Turbo Boost and Hyperthreading disabled. We compiled our C reference implementation with `gcc-4.9.2` and flags `-O3 -fomit-frame-pointer -march=corei7-avx -msse2avx`. We compiled our optimized AVX implementation with `clang-3.5` and flags `-O3 -fomit-frame-pointer -march=native`.

As described in Section 7, the sampling of \mathbf{a} is not running in constant time; we report the median running time and (in parentheses) the average running time for this generation, the server-side key-pair generation and client-side shared-key computation; both over 1000 runs. For all other routines we report the median of 1000 runs. We built the software from [20] on the same machine as ours and—like the authors of [20]—used `openssl speed` for benchmarking their software and converted the reported results to approximate cycle counts as given in Table 2.

Comparison with BCNS and RSA/ECDH. As previously mentioned, the BCNS implementation [20] also uses the dimension $n = 1024$ but the larger modulus $q = 2^{32} - 1$ and the Gaussian error distribution with Gaussian parameter $\sigma = 8/\sqrt{2\pi} = 3.192$. When the authors of BCNS integrated their implementation into SSL it only incurred a slowdown by a factor of 1.27 compared to ECDH when using ECDSA signatures and a factor of 1.08 when using RSA signatures with respect to the number of connections that could be handled by the server. As a reference, the reported cycle counts in [20] for a `nistp256` ECDH on the client side are 2 160 000 cycles (0.8 ms @2.77 GHz) and on the server side 3 221 288 cycles (1.4 ms @2.33 GHz). These numbers are obviously not state of the art for ECDH software. Even on the `nistp256` curve, which is known to be a far-from-optimal choice, it is possible to achieve cycle counts of less than 300 000 cycles for a variable-basepoint scalar multiplication on an Intel Haswell [44]. Also OpenSSL optionally includes fast software for `nistp256` ECDH by Kasper and Langley and we assume that the authors of [20] omitted enabling it. Compared to BCNS, our C implementation is more than 8 times faster and our AVX implementation even achieves a speedup factor of more than 27. At this performance it is in the same ballpark as state-of-the-art ECDH software, even when TLS switches to faster 128-bit secure ECDH key exchange

Table 2: Intel Haswell cycle counts of our proposal as compared to the BCNS proposal from [20].

	BCNS [20]	Ours (C ref)	Ours (AVX2)
Generation of \mathbf{a}		43440 ^a (43607) ^a	37470 ^a (36863) ^a
NTT		55360	8448
NTT^{-1}		59864 ^b	9464 ^b
Sampling of a noise polynomial		32684 ^c	5900 ^c
HelpRec		14608	3404
Rec		10092	2804
Key generation (server)	≈ 2477958	258246 (258965)	88920 (89079)
Key gen + shared key (client)	≈ 3995977	384994 (385146)	110986 (111169)
Shared key (server)	≈ 481937	86280	19422

^a Includes reading a seed from `/dev/urandom`

^b Includes one bit reversal

^c Excludes reading a seed from `/dev/urandom`, which is shared across multiple calls to the noise generation

based on Curve25519 [11], as recently specified in RFC 7748 [59].

In comparison to the BCNS proposal we see a large performance advantage from switching to the binomial error distribution. The BCNS software uses a large pre-computed table to sample from a discrete Gaussian distribution with a high precision. This approach takes 1042700 cycles to samples one polynomial in constant time. Our C implementation requires only 32684 cycles to sample from the binomial distribution. Another factor is that we use the NTT in combination with a smaller modulus. Polynomial multiplication in [20] is using Nussbaumer’s symbolic approach based on recursive negacyclic convolutions [76]. The implementation in [20] only achieves a performance of 342800 cycles for a constant-time multiplication. Additionally, the authors of [20] did not perform pre-transformation of constants (e.g., \mathbf{a}) or transmission of coefficients in FFT/Nussbaumer representation.

Follow-Up Work. We would like to refer the reader to follow-up work in which improvements to NEWHOPE and its implementation were proposed based on a preprint version of this work [4]. In [45] Gueron and Schlieker introduce faster pseudorandom bytes generation by changing the underlying functions, a method to decrease the rejection rate during sampling (see Section 7.1), and a vectorization of the sampling step. Longa and Naehrig [64] optimize the NTT and present new modular reduction techniques and are able to achieve a speedup of factor-1.90 for the C implementation and a factor-1.25 for the AVX implementation compared to the preprint [4] (note that this version has updated numbers).

An alternative NTRU-based proposal and implementation of a lattice-based public-key encryption scheme, which could also be used for key exchange, is given by Bernstein, Chuengsatiansup, Lange, and van Vredendaal in [15], but we leave a detailed comparison to future work. An efficient authenticated lattice-based key exchange scheme has recently been proposed by del Pino, Lyubashevsky, and Pointcheval in [30].

References

- [1] ADRIAN, D., BHARGAVAN, K., DURUMERIC, Z., GAUDRY, P., GREEN, M., HALDERMAN, J. A., HENINGER, N., SPRINGALL, D., THOMÉ, E., VALENTE, L., VANDERSLOOT, B., WUSTROW, E., BÉGUELIN, S. Z., AND ZIMMERMANN, P. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *CCS ’15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 5–17. <https://weakdh.org/>. 4, 5
- [2] ALBRECHT, M., BAI, S., AND DUCAS, L. A subfield lattice attack on overstretched NTRU assumptions. IACR Cryptology ePrint Archive report 2016/127, 2016. <http://eprint.iacr.org/2016/127>. 8, 10
- [3] ALBRECHT, M. R., PLAYER, R., AND SCOTT, S. On the concrete hardness of learning with errors. IACR Cryptology ePrint Archive report 2015/046, 2015. <http://eprint.iacr.org/2015/046/>. 8, 9
- [4] ALKIM, E., DUCAS, L., PÖPPELMANN, T., AND SCHWABE, P. Post-quantum key exchange - a new hope. IACR Cryptology ePrint Archive report 2015/1092, 2015. <https://eprint.iacr.org/2015/1092/20160329:201913>. 14
- [5] ANGEL, Y. Post-quantum secure hybrid handshake based on NewHope. Posting to the tor-dev mailing list, 2016. <https://lists.torproject.org/pipermail/tor-dev/2016-May/010896.html>. 10
- [6] ARORA, S., AND GE, R. New algorithms for learning in presence of errors. In *Automata, Languages and Programming*

- (2011), L. Aceto, M. Henzinger, and J. Sgall, Eds., vol. 6755 of *LNCS*, Springer, pp. 403–415. <https://www.cs.duke.edu/~rongge/LPSN.pdf>. 8
- [7] BABAI, L. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica* 6, 1 (1986), 1–13. <http://www.csie.nuk.edu.tw/~cchen/Lattices/On%20lovasz%20lattice%20reduction%20and%20the%20nearest%20lattice%20point%20problem.pdf>. 5
- [8] BAI, S., AND GALBRAITH, S. D. An improved compression technique for signatures based on learning with errors. In *Topics in Cryptology – CT-RSA 2014* (2014), J. Benaloh, Ed., vol. 8366 of *LNCS*, Springer, pp. 28–47. <https://eprint.iacr.org/2013/838/>. 1
- [9] BARRETT, P. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology – CRYPTO '86* (1987), A. M. Odlyzko, Ed., vol. 263 of *Lecture Notes in Computer Science*, Springer-Verlag Berlin Heidelberg, pp. 311–323. 11
- [10] BECKER, A., DUCAS, L., GAMA, N., AND LAARHOVEN, T. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA '16 Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete Algorithms* (2016 (to appear)), SIAM. 8
- [11] BERNSTEIN, D. J. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography – PKC 2006* (2006), M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958 of *LNCS*, Springer, pp. 207–228. <http://cr.yp.to/papers.html#curve25519>. 14
- [12] BERNSTEIN, D. J. ChaCha, a variant of Salsa20. In *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers* (2008). <http://cr.yp.to/papers.html#chacha>. 12
- [13] BERNSTEIN, D. J., CHOU, T., CHUENGATIANSUP, C., HÜLSING, A., LANGE, T., NIEDERHAGEN, R., AND VAN VREDENDAAL, C. How to manipulate curve standards: a white paper for the black hat. IACR Cryptology ePrint Archive report 2014/571, 2014. <http://eprint.iacr.org/2014/571/>. 4
- [14] BERNSTEIN, D. J., CHUENGATIANSUP, C., LANGE, T., AND SCHWABE, P. Kummer strikes back: new DH speed records. In *Advances in Cryptology – EUROCRYPT 2015* (2014), T. Iwata and P. Sarkar, Eds., vol. 8873 of *LNCS*, Springer, pp. 317–337. full version: <http://cryptojedi.org/papers/#kummer>. 13
- [15] BERNSTEIN, D. J., CHUENGATIANSUP, C., LANGE, T., AND VAN VREDENDAAL, C. NTRU Prime. IACR Cryptology ePrint Archive report 2016/461, 2016. <https://eprint.iacr.org/2016/461>. 14
- [16] BERNSTEIN, D. J., HOPWOOD, D., HÜLSING, A., LANGE, T., NIEDERHAGEN, R., PAPACHRISTODOULOU, L., SCHNEIDER, M., SCHWABE, P., AND WILCOX-O'HEARN, Z. SPHINCS: practical stateless hash-based signatures. In *Advances in Cryptology – EUROCRYPT 2015* (2015), E. Oswald and M. Fischlin, Eds., vol. 9056 of *LNCS*, Springer, pp. 368–397. <https://cryptojedi.org/papers/#sphincs>. 3, 13
- [17] BERNSTEIN, D. J., AND LANGE, T. eBACS: ECRYPT benchmarking of cryptographic systems. <http://bench.cr.yp.to> (accessed 2015-10-07). 10, 13
- [18] BERNSTEIN, D. J., SCHWABE, P., AND ASSCHE, G. V. Tweetable FIPS 202, 2015. <http://keccak.noechoon.org/tweetfips202.html> (accessed 2016-03-21). 12
- [19] BERTONI, G., DAEMEN, J., PEETERS, M., AND ASSCHE, G. V. Keccak. In *Advances in Cryptology – EUROCRYPT 2013* (2013), T. Johansson and P. Q. Nguyen, Eds., vol. 7881 of *LNCS*, Springer, pp. 313–314. 5
- [20] BOS, J. W., COSTELLO, C., NAEHRIG, M., AND STEBILA, D. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy* (2015), pp. 553–570. <http://eprint.iacr.org/2014/599>. 1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
- [21] BRAKERSKI, Z., LANGLOIS, A., PEIKERT, C., REGEV, O., AND STEHLÉ, D. Classical hardness of learning with errors. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing* (2013), ACM, pp. 575–584. <http://arxiv.org/pdf/1306.0281.pdf>
- [22] BURDGES, J. Post-quantum secure hybrid handshake based on NewHope. Posting to the tor-dev mailing list, 2016. <https://lists.torproject.org/pipermail/tor-dev/2016-May/010886.html>. 11
- [23] CADÉ, D., PUJOL, X., AND STEHLÉ, D. fplll 4.0.4, 2013. <https://github.com/dstehle/fplll> (accessed 2015-10-13). 8
- [24] CHECKOWAY, S., FREDRIKSON, M., NIEDERHAGEN, R., EVERSPAUGH, A., GREEN, M., LANGE, T., RISTENPART, T., BERNSTEIN, D. J., MASKIEWICZ, J., AND SHACHAM, H. On the practical exploitability of Dual EC in TLS implementations. In *Proceedings of the 23rd USENIX security symposium* (2014). <https://projectbullrun.org/dual-ec/index.html>. 4
- [25] CHEN, Y. *Lattice reduction and concrete security of fully homomorphic encryption*. PhD thesis, l’Université Paris Diderot, 2013. Available at <http://www.di.ens.fr/~ychen/research/thesis.pdf>. 8, 9
- [26] CHEN, Y., AND NGUYEN, P. Q. BKZ 2.0: Better lattice security estimates. In *Advances in Cryptology – ASIACRYPT 2011*, D. H. Lee and X. Wang, Eds., vol. 7073 of *LNCS*, Springer, 2011, pp. 1–20. <http://www.iacr.org/archive/asiacrypt2011/70730001/70730001.pdf>. 8
- [27] CHU, E., AND GEORGE, A. *Inside the FFT Black Box Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press, Boca Raton, FL, USA, 2000. 11
- [28] COWIE, J., DODSON, B., ELKENBRACHT-HUIZING, R. M., LENSTRA, A. K., MONTGOMERY, P. L., AND ZAYER, J. A world wide number field sieve factoring record: on to 512 bits. In *Advances in Cryptology – ASIACRYPT'96* (1996), K. Kim and T. Matsumoto, Eds., vol. 1163 of *LNCS*, Springer, pp. 382–394. <http://oai.cwi.nl/oai/asset/1940/1940A.pdf>. 8
- [29] DE CLERCQ, R., ROY, S. S., VERCAUTEREN, F., AND VERBAUWHEDE, I. Efficient software implementation of ring-LWE encryption. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2015* (2015), EDA Consortium, pp. 339–344. <http://eprint.iacr.org/2014/725>. 1, 4, 5, 7, 10
- [30] DEL PINO, R., LYUBASHEVSKY, V., AND POINTCHEVAL, D. The whole is less than the sum of its parts: Constructing more efficient lattice-based AKEs. IACR Cryptology ePrint Archive report 2016/435, 2016. <https://eprint.iacr.org/2016/435>. 14
- [31] DUCAS, L., DURMUS, A., LEPOINT, T., AND LYUBASHEVSKY, V. Lattice signatures and bimodal Gaussians. In *Advances in Cryptology – CRYPTO 2013* (2013), R. Canetti and J. A. Garay, Eds., vol. 8042 of *LNCS*, Springer, pp. 40–56. <https://eprint.iacr.org/2013/383/>. 1, 4
- [32] FLUHRER, S. Cryptanalysis of ring-LWE based key exchange with key share reuse. IACR Cryptology ePrint Archive report 2016/085, 2016. <http://eprint.iacr.org/2016/085>. 4
- [33] FUJIOKA, A., SUZUKI, K., XAGAWA, K., AND YONEYAMA, K. Practical and post-quantum authenticated key exchange from one-way secure key encapsulation mechanism. In *Symposium on Information, Computer and Communications Security, ASIA CCS 2013* (2013), K. Chen, Q. Xie, W. Qiu, N. Li, and W. Tzeng, Eds., ACM, pp. 83–94. 3

- [34] GALBRAITH, S. D. Space-efficient variants of cryptosystems based on learning with errors, 2013. <https://www.math.auckland.ac.nz/~sgal018/compact-LWE.pdf>. 5
- [35] GAMA, N., NGUYEN, P. Q., AND REGEV, O. Lattice enumeration using extreme pruning. In *Advances in Cryptology – EUROCRYPT 2010* (2010), H. Gilbert, Ed., vol. 6110 of *LNCS*, Springer, pp. 257–278. <http://www.iacr.org/archive/eurocrypt2010/66320257/66320257.pdf>. 8
- [36] GARG, S., GENTRY, C., AND HALEVI, S. Candidate multilinear maps from ideal lattices. In *Advances in Cryptology – EUROCRYPT 2013* (2013), vol. 7881, Springer, pp. 1–17. <https://eprint.iacr.org/2012/610.pdf>. 8
- [37] GENTLEMAN, W. M., AND SANDE, G. Fast Fourier transforms: for fun and profit. In *Fall Joint Computer Conference* (1966), vol. 29 of *AFIPS Proceedings*, pp. 563–578. http://cis.rit.edu/class/simg716/FFT_Fun_Profit.pdf. 11
- [38] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *STOC '09 Proceedings of the forty-first annual ACM symposium on Theory of computing* (2009), ACM, pp. 169–178. <https://www.cs.cmu.edu/~odonnell/hits09/gentry-homomorphic-encryption.pdf>. 1
- [39] GENTRY, C., PEIKERT, C., AND VAIKUNTANATHAN, V. Trapdoors for hard lattices and new cryptographic constructions. In *STOC '08 Proceedings of the fortieth annual ACM symposium on Theory of computing* (2008), ACM, pp. 197–206. <https://eprint.iacr.org/2007/432.pdf>. 6
- [40] GENTRY, C., AND SZYDŁO, M. Cryptanalysis of the revised NTRU signature scheme. In *Advances in Cryptology – EUROCRYPT 2002* (2002), XXX, Ed., vol. XXX of *LNCS*, Springer, pp. 299–320. https://www.iacr.org/archive/eurocrypt2002/23320295/nsssign_short3.pdf. 8
- [41] GHOSH, S., AND KATE, A. Post-quantum secure onion routing (future anonymity in today's budget). IACR Cryptology ePrint Archive report 2015/008, 2015. <http://eprint.iacr.org/2015/008.pdf>. 3
- [42] GÖTTERT, N., FELLER, T., SCHNEIDER, M., BUCHMANN, J. A., AND HUSS, S. A. On the design of hardware building blocks for modern lattice-based encryption schemes. In *Cryptographic Hardware and Embedded Systems - CHES 2012* (2012), E. Prouff and P. Schaumont, Eds., vol. 7428 of *LNCS*, Springer, pp. 512–529. <http://www.iacr.org/archive/ches2012/74280511/74280511.pdf>. 7, 8, 9, 10
- [43] GUERON, S. Parallelized hashing via j-lanes and j-pointers tree modes, with applications to SHA-256. IACR Cryptology ePrint Archive report 2014/170, 2014. <https://eprint.iacr.org/2014/170.pdf>. 13
- [44] GUERON, S., AND KRASNOV, V. Fast prime field elliptic-curve cryptography with 256-bit primes. *Journal of Cryptographic Engineering* 5, 2 (2014), 141–151. <https://eprint.iacr.org/2013/816.pdf>. 13
- [45] GUERON, S., AND SCHLIEKER, F. Speeding up r-lwe post-quantum key exchange. IACR Cryptology ePrint Archive report 2016/467, 2016. <https://eprint.iacr.org/2016/467.pdf>. 10, 14
- [46] GÜNEYÜSÜ, T., ODER, T., PÖPPELMANN, T., AND SCHWABE, P. Software speed records for lattice-based signatures. In *Post-Quantum Cryptography* (2013), P. Gaborit, Ed., vol. 7932 of *LNCS*, Springer, pp. 67–82. <http://cryptojedi.org/papers/#lattisigns.pdf>. 10, 13
- [47] HANROT, G., PUJOL, X., AND STEHLÉ, D. Terminating BKZ. IACR Cryptology ePrint Archive report 2011/198, 2011. <http://eprint.iacr.org/2011/198.pdf>. 8
- [48] HOFFSTEIN, J., PIPHER, J., SCHANCK, J. M., SILVERMAN, J. H., AND WHYTE, W. Practical signatures from the partial Fourier recovery problem. In *Applied Cryptography and Network Security* (2014), I. Boureanu, P. Owsarski, and S. Vaudey, Eds., vol. 8479 of *LNCS*, Springer, pp. 476–493. <https://eprint.iacr.org/2013/757.pdf>. 1
- [49] HOFFSTEIN, J., PIPHER, J., SCHANCK, J. M., SILVERMAN, J. H., WHYTE, W., AND ZHANG, Z. Choosing parameters for NTRUEncrypt. IACR Cryptology ePrint Archive report 2015/708, 2015. <http://eprint.iacr.org/2015/708.pdf>. 10
- [50] HOFFSTEIN, J., PIPHER, J., AND SILVERMAN, J. H. NTRU: a ring-based public key cryptosystem. In *Algorithmic number theory* (1998), J. P. Buhler, Ed., vol. 1423 of *LNCS*, Springer, pp. 267–288. <https://www.securityinnovation.com/uploads/Crypto/ANTS97.ps.gz>. 1, 4
- [51] HOWGRAVE-GRAHAM, N. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In *Advances in Cryptology-CRYPTO 2007*. Springer, 2007, pp. 150–169. <http://www.iacr.org/archive/crypto2007/46220150/46220150.pdf>. 10
- [52] JINTAI DING, XIANG XIE, X. L. A simple provably secure key exchange scheme based on the learning with errors problem. IACR Cryptology ePrint Archive report 2012/688, 2012. <http://eprint.iacr.org/2012/688.pdf>. 1
- [53] KIRCHNER, P., AND FOUCHE, P.-A. An improved BKW algorithm for LWE with applications to cryptography and lattices. In *Advances in Cryptology – CRYPTO 2015* (2015), R. Gennaro and M. Robshaw, Eds., vol. 9215 of *LNCS*, Springer, pp. 43–62. <https://eprint.iacr.org/2015/552.pdf>. 8
- [54] LAARHOVEN, T. *Search problems in cryptography*. PhD thesis, Eindhoven University of Technology, 2015. <http://www.thijs.com/docs/phd-final.pdf>. 8
- [55] LAARHOVEN, T. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *Advances in Cryptology – CRYPTO 2015* (2015), R. Gennaro and M. Robshaw, Eds., vol. 9216 of *LNCS*, Springer, pp. 3–22. <https://eprint.iacr.org/2014/744.pdf>. 8
- [56] LAARHOVEN, T., MOSCA, M., AND VAN DE POL, J. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography* 77, 2 (2015), 375–400. <https://eprint.iacr.org/2014/907.pdf>. 8
- [57] LANGLEY, A. TLS symmetric crypto. Blog post on imperialviolet.org, 2014. <https://www.imperialviolet.org/2014/02/27/tlssymmetriccrypto.html> (accessed 2015-10-07). 12
- [58] LANGLEY, A., AND CHANG, W.-T. ChaCha20 and Poly1305 based cipher suites for TLS: Internet draft. <https://tools.ietf.org/html/draft-agl-tls-chacha20poly1305-04> (accessed 2015-02-01). 12
- [59] LANGLEY, A., HAMBURG, M., AND TURNER, S. RFC 7748: Elliptic curves for security, 2016. <https://www.rfc-editor.org/rfc/rfc7748.txt>. 14
- [60] LENSTRA, H. W., AND SILVERBERG, A. Revisiting the gentry-szydlo algorithm. In *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds., vol. 8616 of *LNCS*. Springer, 2014, pp. 280–296. <https://eprint.iacr.org/2014/430.pdf>. 8
- [61] LEYS, J., GHYS, E., AND ALVAREZ, A. Dimensions, 2010. <http://www.dimensions-math.org/> (accessed 2015-10-19). 7
- [62] LINDNER, R., AND PEIKERT, C. Better key sizes (and attacks) for LWE-based encryption. In *Topics in Cryptology - CT-RSA 2011* (2011), A. Kiayias, Ed., vol. 6558 of *LNCS*, Springer, pp. 319–339. <https://eprint.iacr.org/2010/613.pdf>. 3

- [63] LIU, Z., SEO, H., ROY, S. S., GROSSCHÄDL, J., KIM, H., AND VERBAUWHEDE, I. Efficient Ring-LWE encryption on 8-bit AVR processors. In *Cryptographic Hardware and Embedded Systems - CHES 2015* (2015), T. Güneysu and H. Handschuh, Eds., vol. 9293 of *LNCS*, Springer, pp. 663–682. <https://eprint.iacr.org/2015/410/>. 4, 7, 10
- [64] LONGA, P., AND NAEHRIG, M. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. IACR Cryptology ePrint Archive report 2016/504, 2016. <https://eprint.iacr.org/2016/504>. 14
- [65] LYUBASHEVSKY, V. Lattice signatures without trapdoors. In *Advances in Cryptology – EUROCRYPT 2012* (2012), D. Pointcheval and T. Johansson, Eds., vol. 7237 of *LNCS*, Springer, pp. 738–755. <https://eprint.iacr.org/2011/537/>. 6
- [66] LYUBASHEVSKY, V., PEIKERT, C., AND REGEV, O. On ideal lattices and learning with errors over rings. In *Advances in Cryptology – EUROCRYPT 2010* (2010), H. Gilbert, Ed., vol. 6110 of *LNCS*, Springer, pp. 1–23. <http://www.di.ens.fr/~lyubash/papers/ringLWE.pdf>. 2
- [67] LYUBASHEVSKY, V., PEIKERT, C., AND REGEV, O. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)* 60, 6 (2013), 43:1–43:35. <http://www.cims.nyu.edu/~regev/papers/ideal-lwe.pdf>. 1, 6
- [68] MELCHOR, C. A., BARRIER, J., FOUSSE, L., AND KILLIJIAN, M. XPIRE: Private information retrieval for everyone. IACR Cryptology ePrint Archive report 2014/1025, 2014. <https://eprint.iacr.org/2014/1025>. 10
- [69] MICCIANCIO, D., AND VOULGARIS, P. Faster exponential time algorithms for the shortest vector problem. In *SODA '10 Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms* (2010), SIAM, pp. 1468–1480. <http://dl.acm.org/citation.cfm?id=1873720.8>
- [70] MONTANARO, A. Quantum walk speedup of backtracking algorithms. arXiv preprint arXiv:1509.02374, 2015. <http://arxiv.org/pdf/1509.02374v2.8>
- [71] MONTGOMERY, P. L. Modular multiplication without trial division. *Mathematics of Computation* 44, 170 (1985), 519–521. <http://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf>. 11
- [72] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>. 5, 10
- [73] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Workshop on cybersecurity in a post-quantum world, 2015. <http://www.nist.gov/itl/csd/ct/post-quantum-crypto-workshop-2015.cfm>. 1
- [74] NATIONAL SECURITY AGENCY. NSA suite B cryptography. https://www.nsa.gov/ia/programs/suiteb_cryptography/, Updated on August 19, 2015. 1
- [75] NGUYEN, P. Q., AND VIDICK, T. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology* 2, 2 (2008), 181–207. <ftp://ftp.di.ens.fr/pub/users/pnguyen/JoMC08.pdf>. 8
- [76] NUSSBAUMER, H. J. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech and Signal Processing* 28, 2 (1980), 205–215. 14
- [77] PEIKERT, C. Lattice cryptography for the Internet. In *Post-Quantum Cryptography* (2014), M. Mosca, Ed., vol. 8772 of *LNCS*, Springer, pp. 197–219. <http://web.eecs.umich.edu/~cpeikert/pubs/suite.pdf>. 1, 2, 3, 4
- [78] PÖPPELMANN, T., DUCAS, L., AND GÜNEYSU, T. Enhanced lattice-based signatures on reconfigurable hardware. In *Cryptographic Hardware and Embedded Systems – CHES 2014* (2014), L. Batina and M. Robshaw, Eds., vol. 8731 of *LNCS*, Springer, pp. 353–370. <https://eprint.iacr.org/2014/254/>. 4
- [79] PÖPPELMANN, T., AND GÜNEYSU, T. Towards practical lattice-based public-key encryption on reconfigurable hardware. In *Selected Areas in Cryptography – SAC 2013* (2013), T. Lange, K. Lauter, and P. Lisoněk, Eds., vol. 8282 of *LNCS*, Springer, pp. 68–85. https://www.ei.rub.de/media/sh_veroeffentlichungen/2013/08/14/lwe_encrypt.pdf. 1, 6, 7, 10, 11
- [80] REGEV, O. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM* 56, 6 (2009), 34. <http://www.cims.nyu.edu/~regev/papers/qcrypto.pdf>. 5
- [81] ROY, S. S., VERCAUTEREN, F., MENTENS, N., CHEN, D. D., AND VERBAUWHEDE, I. Compact Ring-LWE cryptoprocessor. In *Cryptographic Hardware and Embedded Systems – CHES 2014* (2014), L. Batina and M. Robshaw, Eds., vol. 8731 of *LNCS*, Springer, pp. 371–391. <https://eprint.iacr.org/2013/866/>. 4, 5, 11
- [82] SCHNORR, C.-P., AND EUCHNER, M. Lattice basis reduction: improved practical algorithms and solving subset sum problems. *Mathematical programming* 66, 1–3 (1994), 181–199. <http://www.csie.nuk.edu.tw/~cychen/Lattices/Lattice%20Basis%20Reduction%20Improved%20Practical%20Algorithms%20and%20Solving%20Subset%20Sum%20Problems.pdf>. 8
- [83] STEHLÉ, D., AND STEINFELD, R. Making NTRU as secure as worst-case problems over ideal lattices. In *Advances in Cryptology – EUROCRYPT 2011* (2011), K. G. Paterson, Ed., vol. 6632 of *LNCS*, Springer, pp. 27–47. <http://www.iacr.org/archive/eurocrypt2011/66320027/66320027.pdf>. 1, 4
- [84] Tor project: Anonymity online. <https://www.torproject.org/>. 3
- [85] ZHANG, J., ZHANG, Z., DING, J., SNOOK, M., AND DAGDELEN, Ö. Authenticated key exchange from ideal lattices. In *Advances in Cryptology – EUROCRYPT 2015* (2015), E. Oswald and M. Fischlin, Eds., vol. 9057 of *LNCS*, Springer, pp. 719–751. <https://eprint.iacr.org/2014/589/>. 3

Automatically Detecting Error Handling Bugs using Error Specifications

Suman Jana¹, Yuan Kang¹, Samuel Roth², and Baishakhi Ray³

¹Columbia University

²Ohio Northern University

³University of Virginia

Abstract

Incorrect error handling in security-sensitive code often leads to severe security vulnerabilities. Implementing correct error handling is repetitive and tedious especially in languages like C that do not support any exception handling primitives. This makes it very easy for the developers to unwittingly introduce error handling bugs. Moreover, error handling bugs are hard to detect and locate using existing bug-finding techniques because many of these bugs do not display any obviously erroneous behaviors (*e.g.*, crash and assertion failure) but cause subtle inaccuracies.

In this paper, we design, implement, and evaluate EPEX, a tool that uses error specifications to identify and symbolically explore different error paths and reports bugs when any errors are handled incorrectly along these paths. The key insights behind our approach are: (i) real-world programs often handle errors only in a limited number of ways and (ii) most functions have simple and consistent error specifications. This allows us to create a simple oracle that can detect a large class of error handling bugs across a wide range of programs. We evaluated EPEX on 867,000 lines of C Code from four different open-source SSL/TLS libraries (OpenSSL, GnuTLS, mbedTLS, and wolfSSL) and 5 different applications that use SSL/TLS API (Apache httpd, cURL, Wget, LYNX, and Mutt). EPEX discovered 102 new error handling bugs across these programs—at least 53 of which lead to security flaws that break the security guarantees of SSL/TLS. EPEX has a low false positive rate (28 out of 130 reported bugs) as well as a low false negative rate (20 out of 960 reported correct error handling cases).

1 Introduction

Error handling is an important aspect of software development. Errors can occur during a program’s exe-

cution due to various reasons including network packet loss, malformed input, memory allocation failure, *etc.* Handling these errors correctly is crucial for developing secure and robust software. For example, in case of a recoverable error, a developer must ensure that the affected program invokes the appropriate error recovery code. In contrast, if an error is critical, the program must display an appropriate error message and fail in a safe and secure manner. Error handling mistakes not only cause incorrect results, but also often lead to security vulnerabilities with disastrous consequences (*e.g.*, CVE-2014-0092, CVE-2015-0208, CVE-2015-0288, CVE-2015-0285, and CVE-2015-0292). More worryingly, an attacker can often remotely exploit error handling vulnerabilities by sending malformed input, triggering resource allocation failures through denial-of-service attacks *etc.*

To understand how incorrect error handling can lead to severe security vulnerabilities in security-sensitive code, consider the bug in GnuTLS (versions before 3.2.12), a popular Secure Sockets Layer (SSL) and Transport Layer Security (TLS) library used for communicating securely over the Internet, that caused CVE-2014-0092. Listing 1 shows the relevant part of the affected X.509 certificate verification code. The function `_gnutls_verify_certificate2` called another function `check_if_ca` to check whether the issuer of the input certificate is a valid Certificate Authority (CA). `check_if_ca` returns `< 0` to indicate an error (lines 4 and 5 of Listing 1). However, as line 16 shows, `_gnutls_verify_certificate2`, the caller function, only handles the case where the return value is 0 and ignores the cases where `check_if_ca` returns negative numbers as errors. This missing error check makes all applications using GnuTLS incorrectly classify an invalid certificate issuer as valid. This bug completely breaks the security guarantees of all SSL/TLS connections setup using GnuTLS and makes them vulnerable to man-in-the-middle attacks. In summary,

Listing 1: GnuTLS error handling bug (CVE-2014-0092). The lines marked in color gray show an error path and the red lines highlight the source of error handling bug.

```

1 int check_if_ca (...)

2 { ...
3     result = ...;
4     if (result < 0) {
5         goto cleanup;
6     }
7     ...
8     result = 0;
9 }

10 cleanup:
11     return result;
12 }

13 int _gnutls_verify_certificate2 (...)

14 { ...
15     if (check_if_ca (...) == 0) {
16         result = 0;
17         goto cleanup;
18     }
19     ...
20     result = 1;
21 }

22 cleanup:
23     return result;
24 }
```

this bug renders all protections provided by GnuTLS useless.

Developers often introduce error handling bugs unwittingly, as adding error checking code is repetitive and cumbersome (especially in languages like C that do not provide any exception handling primitives). Moreover, a large number of errors in real systems cannot be handled correctly at their source due to data encapsulation and, therefore, must be propagated back to the relevant module. For example, if a protocol implementation receives a malformed packet that cannot be parsed correctly, the parsing error must be appropriately transformed and propagated to the module implementing the protocol state machine in order to ignore the packet and recover gracefully. Implementing correct error propagation in real-world software is non-trivial due to their complex and intertwined code structure.

Automated detection of error handling bugs can help developers significantly improve the security and robustness of critical software. However, there are three major challenges that must be tackled in order to build such a tool: (i) **error path exploration**. Error handling code is only triggered in corner cases that rarely occur during regular execution. This severely limits the ability of dynamic analysis/testing to explore error paths. Moreover, error handling code is usually buried deep inside a program and, therefore, is hard to reach using off-the-shelf symbolic execution tools due to path explosion; (ii) **lack of an error oracle**. Error handling bugs often result

in silent incorrect behavior like producing wrong output, causing memory corruption, *etc.* as shown in the previous example. Therefore, accurately separating incorrect error handling behavior from the correct ones is a hard problem; and (iii) **localizing error handling bugs**. Finally, the effects of error handling bugs are usually manifested far away from their actual sources. Accurately identifying the origin of these bugs is another significant problem.

Our contributions. In this paper, we address all these three problems as discussed below. We design, implement, and evaluate EPEX, a novel algorithm that can automatically detect error-handling bugs in sequential C code.

Identification and scalable exploration of error paths.

As low-level languages like C do not provide any exception handling primitives, the developers are free to use any arbitrary mechanism of their choice for communicating errors. However, we observe that real-world C programs follow simple error protocols for conveying error information across different modules. For example, distinct and non-overlapping integer values are used throughout a C program to indicate erroneous or error-free execution. Integer values like 0 or 1 are often used to communicate error-free execution and negative integers typically indicate errors. Moreover, functions that have related functionality tend to return similar error values. For example, most big number functions in OpenSSL return 0 on error. These observations allow us to create simple error specifications for a given C program, indicating the range of values that a function can return on error. Given such specifications as input, our algorithm performs under-constrained symbolic execution at the corresponding call-sites to symbolically explore only those paths that can return error values and ignores the rest of the paths. Such path filtering minimizes the path exploration problem often plaguing off-the-shelf symbolic execution tools.

Design of an error oracle. We observe that when an error occurs, most C programs usually handle the scenario in one of the following simple ways: (i) propagate an appropriate error value (according to the corresponding error protocol) upstream, (ii) stop the program execution and exit with an error code, or (iii) display/log a relevant error message. We leverage this behavior to create a simple program-independent error oracle. In particular, our algorithm checks whether errors are handled following any of the above three methods along each identified error path; if not, we mark it as a potential bug.

Accurate bug localization. Our error oracle also helps us accurately localize the error handling bugs as it allows our algorithm to detect the bugs at their source. As a side-effect, we can precisely identify buggy error handling code and thus drastically cut down developers' ef-

fort in fixing these bugs.

Implementation and large-scale evaluation. Using our algorithm, we design and implement a tool, EPEX, and evaluate it. EPEX’s analysis is highly parallelizable and scales well in practice. EPEX can be used to find error-handling bugs in any C program as long as the above mentioned assumptions hold true. We evaluated EPEX on a total of 867,000 lines of C code [56] from 4 different open-source SSL/TLS libraries (OpenSSL, GnuTLS, mbedTLS, and wolfSSL) and 5 different applications using SSL/TLS APIs (cURL, Wget, Apache httpd, mutt, and LYNX). EPEX discovered 102 new error handling bugs across these programs—at least 53 of which lead to critical security vulnerabilities that break the security guarantees of SSL/TLS. We also found that EPEX has both low false positive (28 out of 130 reported bugs) and false negative rates (20 out of 960 reported correct error handling cases). Thus, EPEX has a 78% precision and 83% recall on our tested programs. Several of our tested programs (*e.g.*, PolarSSL, cURL, and Apache httpd) have been regularly checked with state-of-the-art static analysis tools like Coverity, Fortify, etc. The fact that none of these bugs were detected by these tools also demonstrates that EPEX can detect bugs that the state-of-the-art bug finding tools miss.

The rest of this paper is organized as follows. We present a brief overview of error handling conventions in C programs in Section 2. We describe our platform- and language-independent technique for detecting error handling bugs in Section 3. The details of implementing our algorithm in Clang and the results are presented in Sections 4 and 5 respectively. We survey the related work in Section 6 and present several directions for future work in Section 7. Section 8 concludes our paper.

2 Error handling in C programs

C does not support exception handling primitives like `try-catch`. In C, a fallible function, which may fail due to different errors, *e.g.*, memory allocation failure or network error, usually communicates errors to the caller function either through return values or by modifying arguments that are passed by reference. While there are no restrictions on the data types/values that can be used to communicate errors, C programmers, in general, create an informal, program-specific error protocol and follow it in all fallible functions of a program to communicate errors. An error protocol consists of a range of error-indicating and non-error-indicating values for different data types. For example, a program may use an error protocol where any negative integer value indicates an error and 0 indicates an error-free execution. Similarly, an error protocol may also use a NULL pointer or a boolean value of `false` to indicate errors. The existence of such

		Error Range	Non-Error Range
Libraries	OpenSSL	$e \leq 0$	$e = 1$
	GnuTLS	$-403 \leq e \leq -1$	$e = 0$
	mbedTLS	$e < 0$	$e = 0$
	wolfSSL	$-213 \leq e \leq -1$	$e \in \{0,1\}$
Applications	httpd [51]	$1 \leq e \leq 720000$	$e = 0$
	curl	$1 \leq e \leq 91$	$e = 0$
	lynx	$-29999 \leq e \leq -1$	$e \geq 0$
	mutt	$e = -1$	$e \geq 0$
	wget	$e = -1$	$e = 0$

e represents the return values of fallible functions

Table 1: Error protocols of the tested libraries/applications

error handling protocols makes it easier for us to create error specifications for different functions of a program.

For example, consider the C programs that we studied in this work. Table 1 shows their error protocols. Fallible functions in OpenSSL usually return 0 or a negative integer to indicate errors and 1 to indicate error-free execution. In contrast, GnuTLS uses negative integers between -1 and -403 to indicate errors and 0 to indicate error-free execution. In spite of the variety of protocols, in all the cases, error-indicating and non-error-indicating ranges for fallible functions do not overlap, to avoid ambiguities.

3 Methodology

In this section, we introduce the details of EPEX (**E**rror **P**ath **E**xplorer), a tool for automatically detecting different types of error handling bugs in sequential C programs. Our key intuition is that if an error is returned by a function in a program path, that error must be handled correctly along that path according to the program’s error convention. Given a function under test, say FT , EPEX identifies possible *error paths*—the paths along which FT returns error values, and ensures that the error values are handled correctly along the error paths at the call site; if not, EPEX reports bugs due to missing error-handling.

3.1 Overview

An overview of EPEX’s workflow for an individual API function is presented in Figure 1. EPEX takes five inputs: the signature of the fallible function under test (FT), the caller functions of FT ($FT_{callers}$), a specification defining a range of error values that FT can return ($FT_{errSpec}$), a range of return values that are used to indicate error-free execution according to the test program’s error protocol ($Global_{nerrSpec}$), and a set of error logging functions used by the program (Loggers). The list of fallible functions,

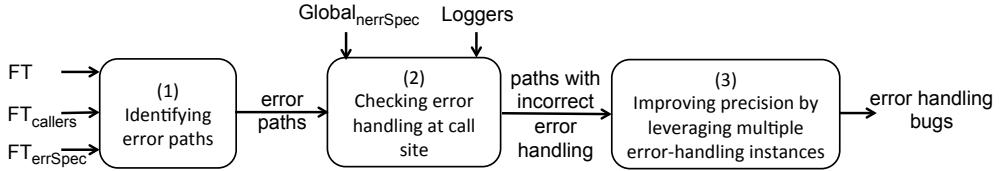


Figure 1: EPEX workflow

their error specifications, and list of error logging functions are created manually, while their caller functions are automatically identified by EPEX. EPEX then works in three steps.

In Step-I, by performing under-constrained symbolic execution at $FT_{callers}$, EPEX identifies the error paths along which FT returns an error value, based on $FT_{errSpec}$. For example, in Listing 1, `check_if_ca`'s error specification says the function will return ≤ 0 on error. Hence, Step-I symbolically executes `_gnutls_verify_certificate2` function and marks the path along the `if` branch in the `check_if_ca` function as an error path (marked in color gray).

Next, in Step-II, EPEX checks if the call site of FT handles the error values correctly. In particular, EPEX checks that if FT returns an error, the error value is handled by the caller in one of the following ways: it (i) pushed the error upstream by returning a correct error value from the caller function, (ii) stopped the program execution with a non-zero error code, or (iii) logged the error by calling a program-specific logging function. If none of these actions take place in an error path, EPEX reports an error handling bug. For instance, in case of Listing 1, the error path returns < 0 at the call site, `_gnutls_verify_certificate2` (line 16). However, the error value is not handled at the call site; in fact it is reset to 1 (line 21), which is a non-error value as per $Global_{nerrSpec}$. Thus, in this case, an error path will return a non-error value. EPEX reports such cases as the potential error-handling bugs (marked in red).

Finally, in Step-III, EPEX checks how error handling code is implemented in other call sites of FT. For example, if all other FT call sites ignore an error value, EPEX does not report a bug even if the error value is not handled properly at the call site under investigation. As $FT_{errSpec}$ may be buggy or symbolic execution engines may be imprecise, this step helps EPEX reduce false positives. The final output of EPEX is a set of program error paths—FT signature, call-site location, and error paths in the caller functions along with EPEX's diagnosis of correct and buggy error handling. We present the detailed algorithm in the rest of this section.

Algorithm 1: EPEX workflow

```

1 EPEX (FT, FTerrSpec, FTcallers, GlobalnerrSpec, Loggers)
Input : function FT, error spec FTerrSpec, callers of FT
        FTcallers, global non-error spec GlobalnerrSpec,
        error logging functions Loggers
Output: Bugs
2
3 Bugs  $\Leftarrow \emptyset$ 
4 shouldHandle  $\Leftarrow \text{False}$ 
5 for each caller  $c \in FT_{callers}$  do
6   for each argument  $a \in c.inputArguments$  do
7     |  $a.isSymbolic \Leftarrow \text{True}$ 
8   end
9   for each path  $p \in c.Paths$  do
10    | isErrPath  $\Leftarrow \text{False}$ 
11    | errPts  $\Leftarrow \emptyset$ 
12    | for each program point  $s \in p$  do
13      |   /* Step-I : identifying error paths */
14      |   | if  $s$  calls FT then
15      |   |   | FTret  $\Leftarrow$  symbolic return value of FT
16      |   |   | isErrPath  $\Leftarrow \text{chkIfErrPath}(FT_{ret},$ 
17      |   |   |   | FTerrSpec)
18      |   |   | if isErrPath = True then
19      |   |   |   | errPts  $\Leftarrow errPts \cup s.location$ 
20      |   |   | end
21      |   | if isErrPath = True then
22      |   |   /* Step-II : checking error Handling */
23      |   |   | isHandled  $\Leftarrow \text{chkErrHandling}(s,$ 
24      |   |   |   | GlobalnerrSpec, Loggers)
25      |   |   | if (isHandled = unhandled) or
26      |   |   |   (isHandled = maybe_handled) then
27      |   |   |   | Bugs  $\Leftarrow Bugs \cup (errPts, isHandled)$ 
28      |   |   | end
29      |   | if (isHandled = handled) or (isHandled =
30      |   |   | maybe_handled) then
31      |   |   |   /* Resetting an error path */
32      |   |   |   | isErrPath  $\Leftarrow \text{False}$ 
33      |   |   |   | errPts  $\Leftarrow \emptyset$ 
34      |   |   | end
35      |   | if (isHandled = handled) then
36      |   |   | shouldHandle  $\Leftarrow \text{True}$ 
37      |   | end
38   end
39 end
40 /* Step-III : Leveraging multiple error-handling instances */
41 if shouldHandle then
42   | return Bugs
43 else
44   | return  $\emptyset$ 
45 end

```

3.2 Step-I: identifying error paths

We define *error paths* to be the program paths in which a function call fails and returns error values. To identify the error paths for a function, EPEX first has to know the error values that a function can return; EPEX takes such information as input (see Section 2 for details). Then the program paths along which the function returns the error values are identified as error paths. The call sites of the failed function are treated as error points (errPts in Algorithm 1). For example, in Listing 1, the program path containing an if-branch (highlighted gray) is an error path; line 16 of `_gnutls_verify_certificate2` is an error point along that error path. Note that an error path can have one or more error points. Given a function under test, say FT, and its caller functions FT_{caller} , the goal of Step-I is to explore all possible error paths going through FT and mark the corresponding error points.

Algorithm 2: Step-I: Identifying error paths

```

1 chkIfErrPath (FTret, FTerrSpec)
  Input : FTret, FTerrSpec
  Output: isErrPath, FTret

2
3 if FTret  $\wedge$  FTerrSpec is satisfiable then
4   /* Error path is possible */
5   if FTret  $\wedge$   $\neg$  FTerrSpec is satisfiable then
6     /* Force the error path, if needed */
7     FTret  $\Leftarrow$  FTret  $\wedge$  FTerrSpec
8   return True
9 else
10  /* Error path is impossible */
11  return False
12 end
```

Exploring error paths. First, EPEX performs under-constrained symbolic execution at each caller function in $FT_{callers}$, and monitors each method call to check if FT is called. If EPEX finds such a call, then right after returning from symbolically executing FT, EPEX checks if the path conditions involving the symbolic return value (FT_{ret}) satisfy its error specifications ($FT_{errSpec}$), as mentioned in the input spec (see Algorithm 2), *i.e.* if $FT_{ret} \wedge FT_{errSpec}$ is satisfiable. This helps EPEX identify two main cases:

- **Error path possible.** If $FT_{ret} \wedge FT_{errSpec}$ is satisfiable, the error path is possible. But while continuing to analyze the error path, EPEX must make sure the constraints make the error path inevitable, so it checks if $FT_{ret} \wedge \neg FT_{errSpec}$ is satisfiable, and if so, sets FT_{ret} to $FT_{ret} \wedge FT_{errSpec}$, so that the constraints force the error path to be taken.
- **Error path impossible.** When $FT_{ret} \wedge FT_{errSpec}$ is unsatisfiable, EPEX considers it as not an error path

and stops tracking it any further.

Algorithm 2 illustrates this process. If a path is considered to be an error path, EPEX notes the corresponding call-site locations in the source code as error points and continues tracking the path in Step-II. In Listing 1, the buggy path has `check_if_ca` return a negative value, which means that it is certainly an error path, and the algorithm returns True, without having to further restrict the constraints.

3.3 Step-II: checking error handling

If a path is marked as an error path in Step-I (*isErrPath* = *True* in Algorithm 1), this step checks whether the error is handled properly along the error path in the caller function. As the symbolic execution engine explores different error paths, we propagate the error path state (*e.g.*, *isErrPath*, FT, and errPts) to any new path forked from conditional branches. We let the rest of the symbolic execution proceed normally unless one of the following happens (see Algorithm 3):

At return point. If EPEX encounters a return statement along an error path, it checks whether the error value is pushed upstream. To do that, Step-II takes program-wide specifications for non-error values ($Global_{nerrSpec}$) as input and checks the constraints on the returned variable of FT_{caller} against $Global_{nerrSpec}$ to determine whether FT_{caller} is returning an error value along the error path. If the returned variable can only contain a non-error value, EPEX marks the corresponding path to be *unhandled*; if it may have a non-error value or an error value, EPEX marks it as *maybe_handled*; and if it cannot have any non-error values, EPEX marks the path as *handled*.

Although both *maybe_handled* and *unhandled* indicate potential bugs, we differentiate between them because in Line 27 of Algorithm 1, we no longer count the path as an error path in the case of *maybe_handled*, since we have already found where the error could be handled; the same error value does not have to be checked repeatedly.

At exit point. A call to libc function `exit` (or other related function like `_exit`) ends the execution of a path. In such a case, EPEX checks the constraints on the symbolic argument to the exit function along an error path: if the symbolic argument can have only error or non-error indicating value, EPEX marks the path as *handled* or *unhandled* respectively. If the argument may have both error and non-error indicating values, EPEX marks the path as *maybe_handled*.

At logging point. The global specifications also support providing the names of the program-specific error logging functions (Loggers). In most C programs, errors are logged through special logging or alerting functions.

If an error path calls an error logging function, EPEX marks that path as *handled*.

In Listing 1, `_gnutls_verify_certificate2` sets `result` to the non-error value, 1, in the error path before returning it, so the algorithm classifies the error as *unhandled*.

Algorithm 3: Step-II: Checking error handling

```

1 chkErrHandling ( $s$ ,  $\text{Global}_{\text{nerrSpec}}$ , Loggers)
  Input : program point  $s$ , global non-error spec
            $\text{Global}_{\text{nerrSpec}}$ , error logging functions Loggers
  Output: isHandled
2
3 if ( $s$  is a top-level ret statement) or ( $s$  is a call to “exit”)
   then
4   |  $tval \leftarrow$  symbolic return value/exit argument
5   | if ( $tval \wedge \text{Global}_{\text{nerrSpec}}$  is satisfiable) and ( $tval \wedge \neg \text{Global}_{\text{nerrSpec}}$  is unsatisfiable) then
6   |   | return unhandled
7   | else if ( $tval \wedge \text{Global}_{\text{nerrSpec}}$  is unsatisfiable) and
     |   ( $tval \wedge \neg \text{Global}_{\text{nerrSpec}}$  is satisfiable) then
8   |   | return handled
9   | else if ( $tval \wedge \text{Global}_{\text{nerrSpec}}$  is satisfiable) and ( $tval \wedge \neg \text{Global}_{\text{nerrSpec}}$  is satisfiable) then
10  |   | return maybe_handled
11 else if  $s \in \text{Loggers}$  then
12   |   return handled
13 return not_checked

```

3.4 Step-III: leveraging multiple error-handling instances

As program documentation may be buggy or symbolic execution engines may be imprecise, EPEX compares the analysis results across multiple callers of the function under test (FT) to minimize false positives. Lines 34 – 45 in Algorithm 1 present this step. If EPEX finds that all the callers of FT return *unhandled* or *maybe_handled*, EPEX ignores the corresponding bugs and does not report them. However, if at least one caller sets *isHandled* to *handled*, all the buggy paths marked from Step-II (line 25 in Algorithm 1) will be reported as bugs. The underlying idea behind this step is inspired by the seminal work of Engler *et al.* [16] where deviant behaviors were shown to indicate bugs.

For example, function `gnutls_x509_trust_list_add_trust_file` adds each Certificate Authority (CA) mentioned in the input file to the list of trusted CAs. In an error-free execution, it returns the number of added CAs. It returns a negative number in case of a parsing error. However, in all the 5 instances in GnuTLS where `gnutls_x509_trust_list_add_trust_file` is called, step II indicates that error values are not handled correctly. In such cases, Step III assumes that the error values can be safely ignored and does not

report any bugs. With manual analysis we confirmed that as trusted certificate authorities can be loaded from multiple sources, such errors can indeed be ignored safely.

4 Implementation

EPEX is implemented using the Clang static analysis framework [42] (part of the LLVM project) and its underlying symbolic execution engine. The Clang analyzer core symbolically explores all feasible paths along the control flow graph of an input program and provides a platform for custom, third-party *checkers* to monitor different paths, inspect the constraints of different symbolic values along those paths, and add additional constraints if necessary. A typical checker often looks for violations of different invariants along a path (*e.g.*, division by zero). In case of a violation, the checker reports bugs. We implement EPEX as a checker inside the Clang analyzer. The rest of this section describes how EPEX is implemented as a Clang checker in detail.

Error specifications. EPEX takes a text file containing the per-function error specifications ($\text{FT}_{\text{errSpec}}$), global non-error specification ($\text{Global}_{\text{nerrSpec}}$), and global error specification ($\text{Global}_{\text{errSpec}}$) as input. Listing 2 shows a sample input file. $\text{FT}_{\text{errSpec}}$ contains five parameters: \langle function name, number of arguments, return type, lower bound of error value, upper bound of error value \rangle . The first three parameters define a function signature. The number of arguments and return type (*e.g.*, integer, boolean, *etc.*) help to disambiguate functions with identical names. The last two optional parameters represent a range of error values that the function can return. For example, error specification for function `RAND_bytes` is: \langle `RAND_bytes`, 2, int, $\geq -1, \leq 0$ \rangle (see line 2 in Listing 2). This shows `RAND_bytes` takes 2 input arguments and returns an integer value. The fourth and fifth parameters indicate that error values range from -1 to 0 . Similarly, if a function `foo` takes four arguments and it has a boolean return type where `False` indicates an error, its error spec will be \langle `foo`, 4, bool, `=False` \rangle . We also support specifications for functions returning `NULL` pointers to indicate errors.

Since most functions in a project follow the same global error convention, (*e.g.*, most OpenSSL functions return 0 to indicate error), error specifications can be simplified by providing a global lower and upper bound of errors. If the per-function error spec does not contain any error range, *i.e.* the fourth and fifth parameters are empty, the checker uses $\text{Global}_{\text{errSpec}}$. Otherwise, the function-specific bounds override $\text{Global}_{\text{errSpec}}$. For instance, OpenSSL functions `RAND_bytes` and `RAND_pseudo_bytes` specify custom lower and upper bounds $-1 \leq$ and \leq

Listing 2: Sample error specifications for OpenSSL functions

```

1  /* Per-func error spec */
2  RAND_bytes, 2, int, >=-1, <=0
3  RAND_pseudo_bytes, 2, int, >=-1, <=0
4
5  /* Per-func spec with empty error ranges
6  (global error ranges will be used) */
7  ASN1_INTEGER_set_int64, 2, int
8  ASN1_INTEGER_set, 1, int
9
10 /* Global error spec */
11 __GLOBAL_ERROR_BOUND__, int, =0, NA
12
13 /* Global non-error spec */
14 __GLOBAL_NOERR_VAL__, int, =1
15 __GLOBAL_NOERR_VAL__, ptr, !=NULL

```

0 respectively, and hence global error bounds are not valid for them. Finally, $\text{Global}_{nerr\text{Spec}}$ contains non-error bounds/values for functions with different return types (see lines 14 and 15 of Listing 2). For example, $\langle \text{__GLOBAL_NOERR_VAL}, \text{int}, = 1 \rangle$ indicates that any function with integer return type returns 1 to indicate an error-free execution. Similarly, $\langle \text{__GLOBAL_NOERR_VAL}, \text{ptr}, !=\text{NULL} \rangle$ indicates an error-free execution for the functions returning pointers will result in the return pointer to be non-null. Such $\text{Global}_{nerr\text{Spec}}$ specifications are used to ensure that the return value of the caller function of FT is pushing the errors upstream correctly.

The error specifications for all the functions under test were created manually. Since, a majority of these functions either follow the per-project global error convention or, at least, functions inside same modules have the same error ranges, *e.g.*, all the big number routines in OpenSSL return 0 on error, the overhead of manual spec generation is not very significant. In fact, it took only one man-day to generate error specs for all 256 functions that we have examined. Table 2 shows the number of specified functions for each library, and the number of unique specifications. Except for WolfSSL, where we used more individualized specifications, each library contained no more than 10 unique error specifications, so that we were able to generate 256 specifications out of only 38 unique constraints.

Table 2: Error specification counts

Library	Functions	Unique Specifications
OpenSSL	109	9
GnuTLS	58	3
mbedTLS	46	10
wolfSSL	43	16
Total	256	38

As the same set of library functions are used by multiple applications, the same error specifications can be reused for all such applications. In fact, for our test

applications, we focused on only OpenSSL API functions. We also found that fallible functions that return booleans or pointers, irrespective of the library they belong to, mostly indicate errors by returning `false` and `NULL` respectively. For functions returning integer error codes, we found that the error codes were almost always represented by a macro or enumerated type that is defined in a header file and therefore was very easy to find. Functions that use the same enumerated type/macro tend to follow the same error protocol. We show some sample error specifications for OpenSSL API functions in Listing 2.

Note that once the error specifications for a set of API functions are created manually, testing new applications using the same API is very easy; the user only needs to add application-specific non-error values (*i.e.* the values indicating error-free execution) for each new application.

Identifying error paths. For identifying error paths, as mentioned in Step-I of Section 3, EPEX checker uses the built-in callback method `checkPostCall`. `checkPostCall` is called once the analyzer core finishes tracking each function body. We overrode `checkPostCall` so that it looks for the functions mentioned in the error spec, *i.e.* checks whether the current function's name, number of arguments, and return type match the specification. In case of a match, Algorithm 2 is called to check whether the function's return value satisfies the lower and upper bounds of error values as given in its error spec; if so, the current path is marked as error path.

Checking error handling. We implemented Step-II by extending the `checkPreStmt` callback method for checking the program state before `return` statements and the `checkPreCall` callback for checking the program state before calling exit functions or any program-specific error logging functions as specified in the input spec. Inside `checkPreStmt` callback, EPEX checks whether the symbolic return value satisfies the global non-error spec (see Algorithm 3). A similar check is performed for exit functions inside `checkPreCall`.

To check the satisfiability conditions of Algorithm 2 and Algorithm 3, we use Clang's built-in constraint solver.

Outputs. EPEX can be run on any single source file (say, `foo.c`) using the command `clang -cc1 -analyze -analyzer-checker=EPEX foo.c`. For running it on large projects like OpenSSL, mbedTLS, etc. we used Clang's `scan-build` utility such that EPEX can be run as part of the regular build process. Scan-build overrides different environment variables (*e.g.*, CC, CPP) to force the build system (*e.g.*, `make`) to use a special compiler script that compiles the input source file and invokes EPEX on it. We pass the `-analyze-header` option to the clang analyzer

core to ensure that the functions defined in any included header files are also analyzed.

The output of EPEX contains one line for each analyzed error path, as shown in Table 3. Each line in the output has four components: name of the caller function, call-site of FT, candidate error-handling location (*i.e.* return instruction, exit call or error logging), and EPEX’s diagnosis about whether the error has been handled correctly or not. As each output line represents an error path and multiple error paths may pass through the same call-site, a call-site for a given FT might be repeated in the output. For example, lines 1 and 2 in Table 3 have the same call-site (ssl_lib.c:1836) but their error handling locations are different (ssl_lib.c:1899 and ssl_lib.c:1905 respectively). Note that we implement Step-III as a separate script and execute it on the output of EPEX before producing the final bug report.

Table 3: Sample EPEX output for OpenSSL function under test: RAND_pseudo_bytes.

Caller Function	Call-site	Error Handling Location	Diagnosis
SSL_CTX_new	ssl_lib.c:1836	ssl_lib.c:1899	handled
SSL_CTX_new	ssl_lib.c:1836	ssl_lib.c:1905	unhandled
dtls1_send_netsession_ticket	d1_srvr.c:1683	d1_srvr.c:1736	maybe_handled

5 Results

5.1 Study subjects

To check whether errors are handled correctly in different functions of popular SSL/TLS libraries as well as applications using them, we ran EPEX on four libraries: OpenSSL, GnuTLS, mbedTLS (formerly known as PolarSSL), and wolfSSL (formerly known as cyaSSL), and five applications that use OpenSSL: cURL, mod_ssl of the Apache HTTP server, Lynx, Mutt, and Wget (see Table 4). For the libraries, we primarily focused on the source files implementing core functionality (*e.g.*, src, lib sub-directories, *etc.*) as opposed to the test files, as detecting bugs in the test code may not be a high priority. All the applications but the HTTP server were small enough to run EPEX on the entire program, although it eventually only produced results for the source files that used the OpenSSL library. For Httpd we only checked mod_ssl. The second column of Table 4 shows the modules investigated by EPEX for each tested library.

For each library, we generate a call graph using a tool named `GNU cflow`¹. From the call graph, we choose top functions that are frequently called by other functions within the same library. Note that here we did not distinguish between internal library functions and

library functions exposed to the outer world as APIs. We further filtered out functions based on their return types—functions returning integer, boolean, and pointers are chosen because Clang’s symbolic analysis engine can currently only handle these types. In addition, we only selected those functions that can fail and return at least one error value. For the applications, we tested all the OpenSSL APIs that the applications are using. We found such APIs by simply using `grep`. Further, we only chose those APIs for which documentations are available, and the APIs that could return errors as integers, booleans or pointers. The third column of Table 4 shows the number of functions tested for a studied program.

Table 4: Study subjects

Projects	Modules	#Functions tested	#Call sites	#Error paths
OpenSSL v1.0.1p	ssl, crypto	46	507	3171
GnuTLS v3.3.17.1	src, lib	50	877	3507
mbedTLS v1.3.11	library	37	505	1621
wolfSSL v3.6.0	wolfcrypt, src	20	138	418
curl v7.47.0	all	17	49	2012
httpd v2.4.18	mod_ssl	14	86	4368
lynx v2.8.8	all	3	23	494
mutt v1.4.2.3	all	3	9	5
wget v1.17.1	all	5	13	2409
Total		195	2207	18005

5.2 General findings

We evaluated EPEX on 195 unique program-API function pairs from 2207 call-sites, and covered 18005 error paths (see Table 4). EPEX found 102 new error-handling bugs from 4 SSL/TLS libraries and 5 applications: 48 bugs in OpenSSL, 23 bugs in GnuTLS, 19 bugs in mbedTLS, and 0 bugs in wolfSSL, 2 in cURL, 7 in httpd, 1 in Lynx, 2 in Mutt, and 0 in Wget (see Table 5). We evaluate EPEX’s performance after completion of Step-II and Step-III separately. Since we are using recent versions of real code, and finding all potential bugs in such code is an extremely difficult problem, we do not have a ground truth for bugs against which to compare the reports. Also, EPEX is not designed to detect all types of error handling bugs. For this paper, we define a bug to be any error path whose output behavior is identical to that of a non-error path, *e.g.*, no logging takes place and the same values as in the non-error paths are propagated upwards through all channels. Thus, for counting false positives and negatives, we do not consider bugs due to incomplete handling, for example, where failures are only logged, but the required cleanup is missing. Table 5 presents the detailed result. After Step-II, EPEX reported 154 bugs in the library code and 29 bugs in the application code. After a manual investigation, we found 61 of them to be false positives. Step-III reduced this false positive to 28 out of 130 reported bugs (106 in library and 24 in application code). Thus, overall, EPEX

¹<http://www.gnu.org/software/cflow/>

detected bugs with 84% precision in the library code and 50% precision in the application code with an overall precision of 78%.

In general, measuring false negatives for static analysis tools is non-trivial as it is hard to be confident about the number of bugs present in the code at any given point of time. However, for the sake of completeness, we checked false negatives by randomly selecting 100 cases at the end of Step-II, where EPEX confirmed that error handling was indeed implemented correctly. We did not find any false negatives in any of those examples, *i.e.* we did not find any bugs that were filtered out at Step-II. However, after Step-III’s optimization, among the bugs that did pass Step-II, we found 15 and 5 false negatives in Library and Application code respectively. Thus, the overall recall of EPEX was approximately 83%.

Table 5: Evaluation of EPEX

Step II		Step III		Summary	
Reported Bugs	False +ve	Reported Bugs	False +ve	True Bugs	Precision
Library					
OpenSSL	51	2	50	2	48 0.96
GnuTLS	41	15	25	1	23 0.96
mbedTLS	35	16	21	2	19 0.90
WolfSSL	27	7	10	2	0 0.80
Total	154	40	106	16	90 0.84
Application					
Curl	6	2	4	2	2 0.5
Httpd	13	6	13	6	7 0.53
Lynx	5	2	3	2	1 0.33
Mutt	3	1	3	1	2 0.67
Wget	2	1	1	1	0 0.00
Total	29	12	24	12	12 0.50
Grand Total	183	52	130	28	102 0.78

In general, EPEX performs better for libraries than applications. There are three main reasons behind this: (i) unlike libraries, applications’ error handling behavior is heavily dependent on their configuration parameters. For example, users can configure the applications to ignore certain errors. EPEX currently cannot differentiate between paths that have different values for these configuration parameters; (ii) Applications are more likely to use complex data types (*e.g.*, error code is embedded within an object) for propagating errors than libraries that are not currently supported by EPEX; and (iii) Applications prioritize user experience over internal consistency, so if the error is recoverable, they will attempt to use a fallback non-error value instead. However, none of these are fundamental limitations of our approach. EPEX can be enhanced to support such cases and improve its accuracy for applications too.

In the following section, we discuss the nature of the detected bugs and the vulnerabilities caused by them in detail with code examples from libraries in Section 5.3 and from applications in Section 5.4. All the described

bugs have been reported to the developers, who, for almost all cases, have confirmed and agreed that they should be fixed.

5.3 Bugs in libraries

From the four SSL/TLS libraries that we tested, we describe seven selected examples. They arise due to various reasons including ignoring error codes, missing checks for certain error codes, checking with a wrong value, and propagating incorrect error values upstream. These bugs affect different modules of the SSL/TLS implementations, and at least 42 of them result in critical security vulnerabilities by completely breaking the security guarantees of SSL/TLS, as discussed below.

Incorrect random number generation. EPEX found 21 instances in OpenSSL where callers of the function `RAND_pseudo_bytes` do not implement the error handling correctly. We provide two such examples below. `RAND_pseudo_bytes` returns cryptographically secure pseudo-random bytes of the desired length. An error-free execution of this function is extremely important to OpenSSL as the security guarantees of all cryptographic primitives implemented in OpenSSL depend on the unpredictability of the random numbers that `RAND_pseudo_bytes` returns. The cryptographically secure random numbers, as returned by `RAND_pseudo_bytes`, are used for diverse purposes by different pieces of OpenSSL code, *e.g.*, creating initialization vectors (IVs), non-repeatable nonces, cryptographic keys. In case of a failure, `RAND_pseudo_bytes` returns 0 or -1 to indicate any error that makes the generated random numbers insecure and unsuitable for cryptographic purposes.

Example 1.

```

1 int PEM_ASN1_write_bio(...)
2 {
3     int ret = 0;
4     ...
5     /* Generate a salt */
6     if (RAND_pseudo_bytes(iv, enc->iv_len) <
7         0)
8         goto err;
9     ...
10    ret = 1;
11    err:
12    OPENSSL_cleanse(iv, sizeof(iv));
13    ...
14    return ret;
15 }
```

Example 2.

```

1 int bnrand(...)
2 {
3     int ret = 0;
4     ...
5     if (RAND_pseudo_bytes(buf, bytes) == -1)
6         goto err;
7     ...
8     ret = 1;
9     err:
10    ...
11    return ret;
12 }
```

The above code shows two examples of incorrect error handling at different call-sites of `RAND_pseudo_bytes` in OpenSSL code. In **Example 1**, function `PEM_ASN1_write_bio` checks only if < 0 , but not if $= 0$. In **Example 2**, function `bnrand` only checks for the -1 value but not for the 0 value. `bnrand` is used by all bignum routines, which are in turn used for key generation by many cryptographic implementations like RSA. These bugs completely break the security guarantees of any cryptographic implementations (RSA, AES, *etc.*) and security protocol implementations (*e.g.*, SSL/TLS, SMIME, *etc.*) in OpenSSL that use such buggy code for random number generation. An attacker can leverage these bugs to launch man-in-the-middle attacks on SSL/TLS connections setup using OpenSSL.

The sources of errors in random number generation functions are diverse and depend on the underlying random number generation mechanism (see Listing 3 in the Appendix for a sample random number generation implementation in OpenSSL). For example, an error can occur due to memory allocation failures or module loading errors. Note that some of these failures can be triggered remotely by an attacker through denial-of-service attacks. Thus, if the errors are not handled correctly, an attacker can break the security guarantees of different cryptographic primitives by making them use insecure random numbers. We have received confirmation from OpenSSL developers about these issues.

Incorrect cryptography implementations. Here, we exhibit an example from OpenSSL demonstrating an error handling bug that EPEX found in the implementation of a cryptographic algorithm.

Insecure SRP keys. EPEX found that the function `SRP_Calc_server_key`, which is part of the SRP (Secure Remote Password) module in OpenSSL, contains an error handling bug while calling `BN_mod_exp`, as shown in the code below.

```

1  BIGNUM *SRP_Calc_server_key(BIGNUM *A,
2      BIGNUM *v,
3      BIGNUM *u, BIGNUM *b, BIGNUM *N)
4  {
5      BIGNUM *tmp = NULL, *S = NULL;
6      BN_CTX *bn_ctx;
7      ...
8      if ((bn_ctx = BN_CTX_new()) == NULL ||
9          (tmp = BN_new()) == NULL ||
10         (S = BN_new()) == NULL)
11         goto err;
12     if (!BN_mod_exp(tmp, v, u, N, bn_ctx))
13         goto err;
14     ...
15   err:
16     BN_CTX_free(bn_ctx);
17     BN_clear_free(tmp);
18     return S;
19 }
```

The `BN_mod_exp` function takes four big numbers (arbitrary-precision integers) `tmp`, `v`, `u`, `N`, and a context

`bn_ctx` as input. It then computes `v` raised to the u^{th} power modulo `N` and stores it in `tmp` (*i.e.* $tmp = v^u \% N$). However, `BN_mod_exp` can fail for different reasons including memory allocation failures. It returns 0 to indicate any such error. The call-site of `BN_mod_exp` (line 12), in fact, correctly checks for such an error and jumps to the error handling code at line 15. The error handling code frees the resources and returns `S` (line 18). However, `S` is guaranteed to be not NULL at this point as it has been allocated by calling a `BN_new` function at line 9. This leads `SRP_Calc_server_key` to return an uninitialized big number `S`. Thus, the functions upstream will not know about the error returned by `BN_mod_exp`, as `SRP_Calc_server_key` is supposed to return a NULL pointer in case of an error. This leads to silent data corruption that can be leveraged to break the security guarantees of the SRP protocol.

Incorrect X.509 certificate revocation. Here we cite two examples from mbedTLS and GnuTLS respectively showing different types of incorrect error handling bugs in implementations of two different X509 certificate revocation mechanisms: CRL (Certificate Revocation List) and OCSP (Online Certificate Status Protocol).

CRL parsing discrepancy. In mbedTLS, EPEX found that `x509_crl_get_version`, which retrieves the version of a X509 certificate revocation list, has an error handling bug while calling function `asn1_get_int` (line 7 in the code below). Function `asn1_get_int` reads an integer from an ASN1 file. It returns different negative values to indicate different errors. In case of a malformed CRL (Certificate Revocation List) file, it returns `POLARSSL_ERR ASN1_UNEXPECTED_TAG` error value. In case of such an error, line 9-13 treats the CRL version as 0 (version 1). Thus, mbedTLS parses a malformed CRL file as version 1 certificate. However, other SSL implementations (*e.g.*, OpenSSL) treat these errors differently and parse it as a version 2 certificate. We are currently discussing the exploitability of this inconsistency with the developers.

```

1  int x509_crl_get_version(unsigned char **p,
2                           const unsigned char
3                           *end,
4                           int *ver )
5  {
6      int ret;
7      if((ret = asn1_get_int(p, end, ver))!= 0)
8      {
9          if( ret ==
10             POLARSSL_ERR ASN1_UNEXPECTED_TAG )
11          {
12              *ver = 0;
13              return(0);
14          }
15      }
16      return(POLARSSL_ERR_X509_INVALID_VERSION +
17      ret);
18  }
19  return( 0 );
```

Incorrect OCSP timestamps. GnuTLS function `gnutls_ocsp_resp_get_single` is used to read the timestamp of an Online Certificate Status Protocol (OCSP) message along with other information. EPEX found an error handling bug in it while calling function `asn1_read_value`, as shown in line 5 in the following code. `asn1_read_value` reads the value of an ASN1 tag. It returns an error while failing to read the tag correctly. `gnutls_ocsp_resp_get_single` correctly checks for the error conditions (line 6), but instead of returning an error value, simply sets the `this_update` parameter to `-1`. However, further upstream, in `check_ocsp_response`, which calls the function `gnutls_ocsp_resp_get_single` (line 16), the corresponding variable `vtime` is not checked for an error value; only the return value is checked, but that is a non-error value. Further down the function, at line 22, `vtime` is used to check whether the message is too old. However, in the error path, since `vtime` is set to `-1` from line 7, the left-hand side of the conditional check will always be a positive number. Due to a large value of the variable `now` (representing current time), the conditional check will always be positive, and result in categorizing all messages to be over the OCSP validity threshold irrespective of their actual timestamp. Depending on the configuration of GnuTLS, this may result in ignoring new OCSP responses containing information on recently revoked certificates.

```

1 int
2 gnutls_ocsp_resp_get_single (... , time_t *
3   this_update)
4 {
5   ...
6   ret = asn1_read_value(resp->basicresp,
7     name, ttime, &len);
8   if (ret != ASN1_SUCCESS) {
9     *this_update = (time_t) (-1);
10  }
11  ...
12  return GNUTLS_SUCCESS;
13 }
14 static int
15 check_ocsp_response(... )
16 {
17   ...
18   ret = gnutls_ocsp_resp_get_single(... , &
19     vtime);
20   if (ret < 0) {
21     ...
22     if ((now - vtime >
23       MAX_OCSP_VALIDITY_SECS) {
24     }
25   }
26 }
```

Incorrect protocol implementations. Here we show two examples from OpenSSL where error handling bugs occur in implementations of two different protocols: Secure/Multipurpose Internet Mail Extensions (S/MIME) and Datagram Transport Layer Security (DTLS).

Faulty parsing of X.509 certificates in S/MIME

EPEX found that the OpenSSL function `cms_SignerIdentifier_cert_cmp` does not check the return value returned by function `X509_get_serialNumber`, as shown in the code below. This code is part of the OpenSSL code that handles S/MIME v3.1 mail. Here, the error point (see line 6) is at the call site of `X509_get_serialNumber`, which returns a pointer to the `ASN1_INTEGER` object that contains the serial number of the input `x509` certificate. However, in case of a malformed certificate missing the serial number, `X509_get_serialNumber` returns `NULL` to indicate an error. In this case, the caller function `cms_SignerIdentifier_cert_cmp` does not check for an error and passes the return value directly to `ASN1_INTEGER_cmp`. Thus, the second argument of `ASN1_INTEGER_cmp` (y in line number 12) is set to `NULL`, in the case of an error. At line 16, `ASN1_INTEGER_cmp` tries to read `y->type` and causes a `NULL` pointer dereference and results in a crash. This can be exploited by a remote attacker to cause a denial of service attack by supplying malformed X.509 certificates. This issue was confirmed by the corresponding developers but they believe that that it is up to the application programmer to ensure that the input certificate is properly initialized.

```

1 int cms_SignerIdentifier_cert_cmp(
2   CMS_SignerIdentifier *sid, X509 *cert)
3 {
4   if (sid->type ==
5     CMS_SIGNERINFO_ISSUER_SERIAL) {
6     ...
7     return ASN1_INTEGER_cmp(serialNumber,
8       X509_get_serialNumber(cert));
9   }
10  ...
11  return -1;
12 }
13 int ASN1_INTEGER_cmp(const ASN1_INTEGER *x,
14   const ASN1_INTEGER *y)
15 {
16   int neg = x->type & V ASN1 NEG;
17   /* Compare signs */
18   if (neg != (y->type & V ASN1 NEG)) {
19     ...
20   }
21   ...
22 }
```

Faulty encoding of X.509 certificates in DTLS. EPEX found that the function `dtls1_add_cert_to_buf` that reads a certificate from DTLS² handshake message contains an error handling bug while calling `i2d_X509` (line 8 in the code below). Function `i2d_X509` encodes the input structure pointed to by `x` into DER format. It returns a negative value to indicate an error, otherwise it returns the length of the encoded data. Here, the caller code (line 8) does not check for error cases, and thus gives no indication of whether the read data was valid or not. In case of an error, this will lead to incorrect results and silent data corruption.

²Datagram Transport Layer Security: a protocol in SSL/TLS family

```

1 static int dtls1_add_cert_to_buf(BUF_MEM *
2     buf, unsigned long *l, X509 **x)
3 {
4     int n;
5     unsigned char *p;
6     ...
7     p = (unsigned char *) &(buf->data[*l]);
8     l2n3(n, p);
9     i2d_X509(x, &p);
10    *l += n + 3;
11 }
12 }
```

5.4 Bugs in applications

Beside libraries, we used EPEX to evaluate error handling implementations in application software that use SSL/TLS library APIs. We have performed tests on 5 programs that use the OpenSSL library: cURL³, httpd⁴, Lynx⁵, Mutt⁶, and Wget⁷. Our error specification included 29 OpenSSL APIs that are used by at least one of these applications. As the results show in Table 5, even though EPEX is not as accurate for applications as for libraries, and we had to discard 2 alerts because the callers did not follow the error protocol, it still found 12 real bugs.

In case of applications, unlike libraries, Step-III of EPEX was able to compare error behavior across multiple applications and libraries that use the same API. This allowed us to detect bugs in the cases where an application developer has consistently made error handling mistakes for an API function as long as other applications using the same API function are correctly handling the errors.

In terms of security effects, the bugs that we found range from causing serious security vulnerabilities to denial-of-service attacks. We found 2 bugs in cURL random number generation modules that can be exploited to make cURL vulnerable to man-in-the-middle attacks. We also found 4 bugs in httpd, Mutt, and Lynx that will cause denial-of-service attacks. The other bugs that we found mostly lead to resource leakage. We provide one example of the cURL random number generation bug below.

cURL ignores the return value of the Pseudorandom number generator RAND_bytes. In case of an error, RAND_bytes will return an output buffer with non-random values. In that case cURL will use it for generating SSL session keys and other secrets. Note that a failure in RAND_bytes can be induced by an attacker by launching a denial of service attack and causing memory allocation failures, file descriptor exhaustion, *etc.*

```

1 int Curl_oss1_random(struct SessionHandle *
2     data, unsigned char *entropy,
3     size_t length)
4 {
5     ...
6     RAND_bytes(entropy, curlx_uztosi(length));
7     return 0; /* 0 as in no error */
8 }
```

5.5 Checking for correct error propagation

Besides producing bugs, EPEX also confirms whether a function call's error handling code correctly takes care of all possible errors. Note that EPEX only checks whether error values are propagated upstream but does not check whether other error handling tasks (e.g., freeing up all acquired resources) have been implemented correctly.

The following example shows an instance where EPEX confirmed that the error codes are correctly propagated by the error handling code. This piece of code is from GnuTLS 3.3.17.1 and contains the fix for the CVE-2014-92 vulnerability that we described in the introduction (Listing 1). EPEX confirmed that the fix indeed correctly handles the error case.

Besides fixing the bug, the updated version of the code has also been slightly refactored and reorganized as shown below. Code in **red** highlights the bug, while **green** shows the fix. The return type of function `check_if_ca` has been updated to `bool`, where returning false (0) indicates an error (see line 1 and 10). The caller function `verify_crt` is correctly checking $\neq 1$ (*i.e.* True) at line 17 to handle the error case.

```

1 int bool
2 check_if_ca(...)
3 { ...
4     if (ret < 0) {
5         gnutls_assert();
6         goto fail;
7     }
8
9     fail:
10    result = 0;
11
12    return result;
13 }
14
15 bool verify_crt(...)
16 {
17     if (check_if_ca(...) == 0 != 1) {
18         result = 0;
19         goto cleanup;
20     }
21     ...
22     cleanup:
23     ...
24     return result;
25 }
26 }
```

We also used EPEX to successfully check the fixes for other CVEs mentioned in Section 1 (CVE-2015-0208, CVE-2015-0288, CVE-2015-0285, and CVE-2015-0292).

5.6 Imprecision in EPEX Analysis

The 130 potential bugs reported by EPEX includes 28 false positives and incorrectly excludes 20. In the li-

³<http://curl.haxx.se/>

⁴<https://httpd.apache.org/>

⁵<http://lynx.invisible-island.net/>

⁶<http://www.mutt.org/>

⁷<https://www.gnu.org/software/wget/wget.html>

ibraries, most of the false positives appeared due to the limitations of underlying Clang symbolic analysis engine. The interprocedural analysis supported by Clang’s symbolic analysis engine is currently limited to the functions defined within an input source file or functions included in the file through header files. Therefore, the symbolic analyzer is not able to gather correct path conditions and return values for the functions defined in other source files. For example, in the code below, EPEX reported error since the return value of `X509_get_serialNumber` is not checked at line 5. However, inside the callee, `ASN1_STRING_dup`, the error condition is checked at line 17 and the NULL value is returned. This return value (`serial`) is further checked at line 6. Since, `ASN1_STRING_dup` is implemented in a different file, EPEX could not infer that the `ASN1_STRING_dup` call in line 5 will always return NULL if `X509_get_serialNumber` returns an error. Note that if the pattern of not checking error for `X509_get_serialNumber` calls were consistent across all call-sites, EPEX would not have reported this false positive due to Step-III in Section 3).

```

1 AUTHORITY_KEYID *v2i_AUTHORITY_KEYID(...
2 {
3     ...
4     serial = ASN1_INTEGER_dup(
5         X509_get_serialNumber(cert));
6     if (!isname || !serial) {
7         X509V3err(...);
8         goto err;
9     }
10    ...
11 }
12 ASN1_STRING *ASN1_STRING_dup(
13     const ASN1_STRING *str)
14 {
15     ASN1_STRING *ret;
16     if (!str)
17         return NULL;
18     ...
20 }
```

EPEX performed the worst in wolfSSL, mostly due to confusion arising from compile-time configuration settings affecting the function `mp_init`. EPEX raised 8 alerts for the function, but after contacting the developers, we learned that the corresponding functions can be configured, at compilation time, to be either fallible or infallible. All the reported call sites were only compiled if the functions were configured to be infallible. Therefore, our error specifications should not have marked these functions as fallible. On the other hand, in the applications, the most frequent causes are 5 instances of fallbacks, which are characteristic of applications. Still, missed checks in external functions are the second most frequent cause, at 3 cases. The remaining causes are alternative error propagation channels, and deliberate disregard for the error, due to either a conscious choice of the programmer, or a configuration parameter, as mentioned in Section 5.2.

Given the false positives due to checks by external functions, a natural solution would be to have all functions validate their input. While this is a good practice for library programmers, application programmers should not depend on functions, whose implementation they often do not control, to follow this practice. For debugging purposes, it would appear that the function receiving the invalid return value is at fault. Moreover, not all functions can cleanly handle invalid input. Comparison functions such as `ASN1_INTEGER_cmp` only return non-error values, so the only safe response would be to terminate the program, which is a drastic action that can easily be averted by checking the parameters in the first place.

5.7 Performance analysis

EPEX is integrated with the test project’s building procedure through the Clang framework. We ran all our tests on Linux servers with 4 Intel Xeon 2.67GHz processors and 100 GB of memory. The following table shows the performance numbers. EPEX’s execution time is comparable to that of other built-in, simple checkers in Clang (*e.g.*, division-by-zero) as shown in the table below.

	Regular build	Division-by-zero in-built checker	EPEX checker
wolfSSL	0.05m	3.08m	2.68m
mbedTLS	0.67m	3.72m	2.83m
GnuTLS	1.85m	13.28m	12.82m
OpenSSL	8.25m	186.9m	132.33m
cURL	0.18m	13.96m	12.95m
httpd	0.04m	4.68m	4.51m
Lynx	0.55m	71.35m	71.73m
Mutt	0.10m	13.03m	13.12m
Wget	0.03m	5.63m	5.66m

6 Related work

6.1 Automated detection of error handling bugs

Rubio-González *et al.* [45, 21] detected incorrect error handling code in the Linux file system using a static control and data-flow analysis. Their technique was designed to detect bugs caused by faulty code that either overwrite or ignore error values. In addition to these two cases, we check whether appropriate error values are propagated upstream as per global error protocol of the analyzed program. We use module-specific error specifications as opposed to hard coded error values like `-EIO`, `-ENOMEM`, *etc.* used by Rubio-González *et al.* This helps us in reducing the number of false positives significantly; for instance, unlike [45, 21], we do not report a bug when an error value is over-written by another error value that conforms to the global error protocol. Our usage of symbolic analysis further minimizes false

positives as symbolic analysis, unlike the data-flow analysis used in [45, 21], can distinguish between feasible and infeasible paths.

Acharya *et al.* [1] automatically inferred error handling specifications of APIs by mining static traces of their run-time behaviors. Then, for a different subject system, they found several bugs in error handling code that do not obey the inferred specifications. The static traces were generated by MOPS [11] that handles only control dependencies and minimal data-dependencies. As observed by Acharya *et al.*, lack of extensive data-dependency support (*e.g.*, pointer analysis, aliasing, *etc.*) introduced imprecision in their results. By contrast, our symbolic execution engine with extensive memory modeling support minimizes such issues. Further, to identify error handling code blocks corresponding to an API function, Acharya *et al.* leveraged the presence of conditional checks on the API function’s return value and/or ERNNO flag. They assumed that if such a conditional check leads to a return or exit call, then it is responsible for handling the error case. Such assumption may lead to false positives where conditional checks are performed on non-error cases. Also, as noted by Acharya *et al.*, for functions that can return multiple non-error values, they cannot distinguish them from error cases. By contrast, we create our error specifications from the program documentation and thus they do not suffer from such discrepancies.

Lawall *et al.* [31] used Coccinelle, a program matching and transformation engine, to find missing error checks in OpenSSL. By contrast, we not only look for error checks but also ensure that the error is indeed handled correctly. This allows us to find a significantly larger class of error handling problems. Also, unlike our approach, Lawall *et al.*’s method suffers from an extremely high false positive rate.

Several other approaches to automatically detect error/exception handling bugs have been proposed for Java programs [52, 53, 44, 8, 54]. However, since the error handling mechanism is quite different in Java than C (*e.g.*, the `try-catch-finally` construct is not supported in C), these solutions are not directly applicable to C code.

Static analysis has been used extensively in the past to find missing checks on security critical objects [48, 57, 49]. However, none of these tools can detect missing/incorrect error handling checks. Complementary to our work, and other static approaches, dynamic analysis methods have been developed to discover the practical effects of error handling bugs, although they do so at the cost of lower coverage of error paths, as well as unknown failure modes. Fault injection frameworks such as LFI bypass the problem of the unlikelihood of errors by injecting failures directly into fallible functions.

LFI includes a module for automatically inferring error specifications, although it is not usable in our case, since static analysis requires explicitly identifying error and non-error values, and not just differentiate between them [34].

6.2 Symbolic execution

The idea of symbolic execution was initially proposed by King *et al.* [29]. Concolic execution is a recent variant of symbolic execution where concrete inputs guide the execution [19, 10, 47]. Such techniques have been used in several recent projects for automatically finding security bugs [27, 46, 22, 20].

KLEE [9], by Cadar *et al.*, is a symbolic execution engine that has been successfully used to find several bugs in UNIX coreutils automatically. UC-KLEE [40], which integrates KLEE and lazy initialization [26], applies more comprehensive symbolic execution over a bounded exhaustive execution space to check for code equivalence; UC-KLEE has also been effective in finding bugs in different tools, including itself. Recently, Ramos *et al.* applied UC-KLEE to find two denial-of-service vulnerabilities in OpenSSL [41].

SAGE, by Godefroid *et al.* [20], uses a given set of inputs as seeds, builds symbolic path conditions by monitoring their execution paths, and systematically negates these path conditions to explore their neighboring paths, and generate input for fuzzing. SAGE has been successfully used to find several bugs (including security bugs) in different Windows applications like media players and image processors. SAGE also checks for error handling bugs, but only errors from user inputs, and not environmental failures, which are unlikely to appear when only user input is fuzzed.

Ardilla, by Kiezun *et al.* [27], automates testing of Web applications for SQL injection and cross-site scripting attacks by generating test inputs using dynamic taint analysis that leverages concolic execution and mutates the inputs using a library of attack patterns.

Existing symbolic execution tools are not well suited for finding error handling bugs for two primary reasons: (i) The existing symbolic execution tools depend on obvious faulty behaviors like crashes, assertion failures, *etc.* for detecting bugs. A large number of error handling bugs are completely silent and do not exhibit any such behavior. (ii) As the number of paths through any reasonable sized program is very large, all symbolic execution tools can only explore a fraction of those paths. The effects of most non-silent error handling bugs show up much further downstream from their source. An off-the-shelf symbolic execution tool can only detect such cases if it reaches that point. By contrast, our algorithm for identifying and exploring error paths enables EPEX

to detect completely silent and non-silent error handling bugs at their sources. This makes it easy for the developers to understand and fix these bugs.

6.3 Security of SSL/TLS implementations

Several security vulnerabilities have been found over the years in both SSL/TLS implementations and protocol specifications [15, 43, 2, 5, 7, 4, 3]. We briefly summarize some of these issues below. A detailed survey of SSL/TLS vulnerabilities can be found in [13].

Multiple vulnerabilities in certification validation implementations, a key part of the SSL/TLS protocol, were reported by Moxie Marlinspike [38, 37, 36, 35]. Similar bugs have been recently discovered in the SSL implementation on Apple iOS [24]. Another certificate validation bug (“goto fail”) was reported in Mac OS and iOS [30] due to an extra goto statement in the implementation of the SSL/TLS handshake protocol. The affected code did not ensure that the key used to sign the server’s key exchange matches the key in the certificate presented by the server. This flaw made the SSL/TLS implementations in MacOS and iOS vulnerable to active Man-In-The-Middle (MITM) attackers. This bug was caused by unintended overlapping of some parts of a non-error path and an error path. However, this is not an error handling bug like the ones we found in this paper.

Hash collisions [50] and certificate parsing discrepancies between certificate authorities (CAs) and Web browsers [25] can trick a CA into issuing a valid certificate with the wrong subject name or even a valid intermediate CA certificate. This allows an attacker to launch a successful MITM attack against any arbitrary SSL/TLS connection.

Georgiev *et al.* [18] showed that incorrect usage of SSL/TLS APIs results in a large number of certificate validation vulnerabilities in different applications. Fahl *et al.* [17] analyzed incorrect SSL/TLS API usage for Android applications. Brubaker *et al.* [6] designed Frankencerts, a mechanism for generating synthetic X.509 certificates based on a set of publicly available seed certificates for testing the certificate validation component of SSL/TLS libraries. They performed differential testing on multiple SSL/TLS libraries using Frankencerts and found several new security vulnerabilities. Chen *et al.* [12] improved the coverage and efficiency of Brubaker *et al.*’s technique by diversifying the seed certificate selection process using Markov Chain Monte Carlo (MCMC) sampling. However, all these techniques are black-box methods that only focus on the certificate validation part of the SSL/TLS implementations. By contrast, our white-box analysis is tailored to look for flawed error handling code in any sequential C code.

Flawed pseudo-random number generation can produce insecure SSL/TLS keys that can be easily compromised [32, 23]. We have also reported several bugs involving pseudo-random number generator functions in this paper, although their origins are completely different, *i.e.*, unlike [32, 23], they are caused by incorrect error handling.

7 Future work

Automated inference of error specifications. One limitation of our current implementation of EPEX is that it requires the input error specifications to be created manually by the user. Automatically generating the error specifications will significantly improve EPEX’s usability. One possible way to automatically infer the error specifications is to identify and compare the path constraints imposed along the error paths (*i.e.*, the paths along which a function can fail and return errors) across different call-sites of the same function. However, in order to do so, the error paths must first be automatically identified. This leads to a chicken-and-egg problem as the current prototype of EPEX uses the input error specifications to identify the error paths.

To solve this problem, we plan to leverage different path features that can distinguish the error paths from non-error paths. For example, error paths are often more likely to return constant values than non-error paths [33]. Error paths are also more likely to call functions like exit (with a non-zero argument) than regular code for early termination. Further, since errors invalidate the rest of the computation, the lengths of the error paths (*i.e.*, number of program statements) might be, on average, shorter than the non-error paths. An interesting direction for future research will be to train a supervised machine learning algorithm like Support Vector Machines (SVMs) [14] for identifying error paths using such different path features. The supervised machine learning algorithm can be trained using a small set of error and non-error paths identified through manually created error specifications. The resulting machine learning model can then be used to automatically identify different error paths and infer error specifications by comparing the corresponding path constraints.

Automatically generating bug fixes. As error-handling code is often repetitive and cumbersome to implement, it might be difficult for developers to keep up with EPEX and fix all the reported bugs manually. Moreover, manual fixes introduced by a developer might also be buggy and thus may introduce new error handling bugs. In order to avoid such issues, we plan to automatically generate candidate patches to fix the error handling bugs reported by EPEX. Several recent projects [55, 39, 28] have successfully generated patches

for fixing different types of bugs. Their main approach is dependent on existing test suites—they first generate candidate patches by modifying existing code and then validate the patches using existing test cases. While this generic approach can be applied in our setting, we cannot use the existing schemes as error handling bugs are, in general, hard to detect through existing test cases. Also, these approaches typically focus on bug fixes involving only one or two lines of code changes. However, the error handling bugs are not necessarily limited to such small fixes. Solving these issues will be an interesting direction for future work.

8 Conclusion

In this paper, we presented EPEX, a new algorithm and a tool that automatically explores error paths and finds error handling bugs in sequential C code. We showed that EPEX can efficiently find error handling bugs in different open-source SSL/TLS libraries and applications with few false positives; many of these detected bugs lead to critical security vulnerabilities. We also demonstrate that EPEX could also be useful to the developers for checking error handling code.

9 Acknowledgments

We would like to thank Ben Livshits, the shepherd of this paper, and the anonymous reviewers whose suggestions have improved the presentation of our work. We would also like to thank David Evans for his feedback on an earlier draft of this paper. This work is sponsored in part by Air Force Office of Scientific Research (AFOSR) grant FA9550-12-1-0162. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFOSR.

References

- [1] M. Acharya and T. Xie. Mining API Error-Handling Specifications from Source Code. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2009.
- [2] N. AlFardan and K. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.
- [3] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.
- [4] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [5] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *International Cryptology Conference (CRYPTO)*, 1996.
- [6] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [7] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security Symposium*, 2003.
- [8] R. Buse and W. Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.
- [9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
- [10] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *International SPIN Workshop on Model Checking of Software (SPIN)*, 2005.
- [11] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security (CCS)*, 2002.
- [12] Y. Chen and Z. Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *ACM SIGSOFT International Symposium on the Foundations of Software (FSE)*, 2015.
- [13] J. Clark and P. van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.
- [14] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
- [15] T. Duong and J. Rizzo. Here come the \oplus ninjas. http://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf, 2011.
- [16] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP)*, 2001.
- [17] S. Fahl, M. Harbach, T. Muders, and M. Smith. Why Eve and Mallory love Android: An analysis of SSI (in)security on Android. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
- [18] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
- [19] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2005.
- [20] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network & Distributed System Security Symposium (NDSS)*, 2008.
- [21] H. Gunawi, C. Rubio-González, A. Arpacı-Dusseau, R. Arpacı-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *USENIX Conference on File and Storage Technologies (FAST)*, 2008.
- [22] W. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.
- [23] N. Heninger, Z. Durumeric, E. Wustrow, and A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, 2012.
- [24] CVE-2011-0228. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0228>, 2011.
- [25] D. Kaminsky, M. Patterson, and L. Sassaman. PKI layer cake: New collision attacks against the global X.509 infrastructure. In

- FC*, 2010.
- [26] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.
- [27] A. Kiezun, P. Guo, K. Jayaraman, and M. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *International Conference on Software Engineering (ICSE)*, 2009.
- [28] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE)*, 2013.
- [29] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [30] A. Langley. Apple’s SSL/TLS bug. <https://www.imperialviolet.org/2014/02/22/applebug.html>, 2014.
- [31] J. Lawall, B. Laurie, R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in openssl using coccinelle. In *European Dependable Computing Conference (EDCC)*, 2010.
- [32] A. Lenstra, J. Hughes, M. Augier, J. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, Whit is right. <http://eprint.iacr.org/2012/064>, 2012.
- [33] P. Marinescu and G. Cadea. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems (TOCS)*, 29(4), 2011.
- [34] P. D. Marinescu, R. Banabic, and G. Cadea. An extensible technique for high-precision testing of recovery code. In *USENIX Annual Technical Conference*, 2010.
- [35] M. Marlinspike. IE SSL vulnerability. <http://www.thoughtcrime.org/ie-ssl-chain.txt>, 2002.
- [36] M. Marlinspike. More tricks for defeating SSL in practice. DEF-CON, 2009.
- [37] M. Marlinspike. New tricks for defeating SSL in practice. Black Hat DC, 2009.
- [38] M. Marlinspike. Null prefix attacks against SSL/TLS certificates. <http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf>, 2009.
- [39] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2015.
- [40] D. Ramos and D. Engler. Practical, low-effort equivalence verification of real code. In *International Conference on Computer-Aided Verification (CAV)*, 2011.
- [41] D. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *USENIX Security Symposium*, 2015.
- [42] Checker developer manual. http://clang-analyzer.llvm.org/checker_dev_manual.html.
- [43] J. Rizzo and T. Duong. The CRIME attack. In *Ekoparty*, 2012.
- [44] M. Robillard and G. Murphy. Analyzing exception flow in Java programs. In *ACM SIGSOFT International Symposium on the Foundations of Software (FSE)*, 1999.
- [45] C. Rubio-González, H. Gunawi, B. Liblit, R. Arpacı-Dusseau, and A. Arpacı-Dusseau. Error propagation analysis for file systems. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2009.
- [46] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.
- [47] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ACM SIGSOFT International Symposium on the Foundations of Software (FSE)*, 2005.
- [48] S. Son, K. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2011.
- [49] V. Srivastava, M. Bond, K. McKinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple API implementations. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2011.
- [50] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. Osvik, and B. Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *International Cryptology Conference (CRYPTO)*, 2009.
- [51] The Apache Software Foundation. Apache portable runtime: Error codes. Available at https://apr.apache.org/docs/apr/1.4/group_apr_errno.html, 2011.
- [52] W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2004.
- [53] W. Weimer and G. Necula. Mining Temporal Specifications for Error Detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.
- [54] W. Weimer and G. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2008.
- [55] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*, 2009.
- [56] D. A. Wheeler. Sloccount. Available at <http://www.dwheeler.com/slocount/>, 2015.
- [57] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: exposing missing checks in source code for vulnerability discovery. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

A Appendix

Listing 3: Sample implementation of RAND_pseudo_bytes in OpenSSL

```

1  /* crypto/engine/hw_aep.c */
2  int aep_rand(unsigned char *buf, int len)
3  {
4      ...
5      AEP_RV rv = AEP_R_OK;
6      AEP_CONNECTION_HNDL hConnection;
7
8      rv = aep_get_connection(&hConnection);
9      if (rv != AEP_R_OK) {
10          AEPHKerr(AEPHK_F_AEP_RAND,
11                  AEPHK_R_GET_HANDLE_FAILED);
12          goto err_nounlock;
13      }
14      if (len > RAND_BLK_SIZE) {
15          rv = p_AEP_GenRandom(hConnection, len,
16                               2, buf, NULL);
17          if (rv != AEP_R_OK) {
18              AEPHKerr(AEPHK_F_AEP_RAND,
19                      AEPHK_R_GET_RANDOM_FAILED);
20              goto err_nounlock;
21          }
22          ...
23      }
24      return 1;
25  err_nounlock:
26      return 0;
27 }
```

Table 6: Tested functions and bug counts

	Function Name	Bug Count	False Positives
OpenSSL	ASN1_INTEGER_set	4	0
	BN_mod_exp	3	0
	BN_sub	2	0
	EC_KEY_up_ref	1	0
	EC_POINT_cmp	1	0
	PEM_read_bio_X509	2	0
	RAND_pseudo_bytes	20	1
	X509_get_serialNumber	3	1
	i2a ASN1_INTEGER	3	0
	i2d_X509	9	0
Total		48	2
GnuTLS	asn1_read_value	4	0
	asn1_write_value	3	0
	gnutls_openpgp_crt_get_subkey_idx	1	0
	gnutls_openpgp_privkey_get_subkey_idx	3	0
	gnutls_privkey_get_pk_algorithm	3	1
	gnutls_x509_crq_get_dn_by_oid	2	0
	gnutls_x509_crq_get_extension_info	1	0
	gnutls_x509_crq_get_pk_algorithm	2	0
	gnutls_x509_crt_get_serial	1	0
	gnutls_x509_privkey_import	0	1
	gnutls_x509_privkey_import_pkcs8	1	0
	record_overhead_rt	2	0
	Total	23	2
mbedTLS	aes_setkey_enc	0	1
	asn1_get_int	2	0
	asn1_get_tag	8	0
	md_hmac_starts	2	0
	md_init_ctx	2	0
	mpi_fill_random	5	0
	ssl_handshake	0	1
Total		19	2
wolfSSL	wc_InitRsaKey	0	1
	wc_ShaHash	0	1
	mp_init	0	8
	Total	0	10
cURL	RAND_bytes	2	0
	SSL_get_peer_cert_chain	0	1
	SSL_shutdown	0	1
	Total	2	2
httpd	BIO_free	4	0
	BIO_new	1	1
	SSL_CTX_new	1	0
	SSL_CTX_use_certificate_chain_file	1	0
	SSL_get_peer_cert_chain	0	1
	SSL_get_peer_certificate	0	1
	SSL_get_verify_result	0	1
	SSL_read	0	1
	SSL_write	0	1
	Total	7	6
Lynx	SSL_set_fd	1	0
	SSL_CTX_new	0	2
	Total	1	2
Mutt	SSL_CTX_new	0	1
	BIO_new	1	0
	SSL_shutdown	1	0
	Total	2	1
Wget	BIO_new	0	1
	Total	0	1
Grand Total		102	28

APISAN: Sanitizing API Usages through Semantic Cross-checking

Insu Yun Changwoo Min Xujie Si Yeongjin Jang Taesoo Kim Mayur Naik
Georgia Institute of Technology

Abstract

API misuse is a well-known source of bugs. Some of them (e.g., incorrect use of SSL API, and integer overflow of memory allocation size) can cause serious security vulnerabilities (e.g., man-in-the-middle (MITM) attack, and privilege escalation). Moreover, modern APIs, which are large, complex, and fast evolving, are error-prone. However, existing techniques to help finding bugs require manual effort by developers (e.g., providing specification or model) or are not scalable to large real-world software comprising millions of lines of code.

In this paper, we present APISAN, a tool that automatically infers correct API usages from source code without manual effort. The key idea in APISAN is to extract likely correct usage patterns in four different aspects (e.g., causal relation, and semantic relation on arguments) by considering semantic constraints. APISAN is tailored to check various properties with security implications. We applied APISAN to 92 million lines of code, including Linux Kernel, and OpenSSL, found 76 previously unknown bugs, and provided patches for all the bugs.

1 Introduction

Today, large and complex software is built with many components integrated using APIs. While APIs encapsulate the internal state of components, they also expose rich semantic information, which renders them challenging to use correctly in practice. Misuse of APIs in turn leads to incorrect results and more critically, can have serious security implications. For example, a misuse of OpenSSL API can result in man-in-the-middle (MITM) attacks [22, 26], and seemingly benign incorrect error handling in Linux (e.g., missing a check on `kmalloc()`) can allow DoS or even privilege escalation attacks [12]. This problem, in fact, is not limited to API usage, but pervades the usage of all functions, which we generally refer to as APIs in this paper.

Many different tools, techniques, and methodologies have been proposed to address the problem of finding or preventing API usage errors. Broadly, all existing techniques either require (1) manual effort—API-specific specifications (e.g., SSL in SSLint [26], setuid [10, 15]), code annotations (e.g., lock operations in Sparse [41]),

correct models (e.g., file system in WOODPECKER [11]), or (2) an accurate analysis of source code [6, 7], which is hard to scale to complex, real-world system software written in C/C++.

We present a fully automated system, called APISAN for finding API usage errors. Unlike traditional approaches that require API-specific specifications or models, APISAN infers the correct usage of an API from other uses of the API, regarding the majority usage pattern as a *semantic belief*, i.e., the likely correct use. Also, instead of relying on whole-program analysis, APISAN represents correct API usage in a probabilistic manner, which makes it scalable beyond tens of millions of lines of low-level system code like the Linux kernel. In APISAN, the higher the observed number of API uses, potentially even from different programs, the stronger is the belief in the inferred correct use. Once APISAN extracts such semantic beliefs, it reports deviations from the beliefs as potential errors together with a probabilistic ranking that reflects their likelihood.

A hallmark of APISAN compared to existing approaches [1, 18, 28, 29] for finding bugs by detecting contradictions in source code is that it achieves precision by considering semantic constraints in API usage patterns. APISAN infers such constraints in the form of symbolic contexts that it computes using a symbolic execution based technique. The technique, called *relaxed symbolic execution*, circumvents the path-explosion problem by limiting exploration to a bounded number of intra-procedural paths that suffice in practice for the purpose of inferring semantic beliefs.

APISAN computes a database of symbolic contexts from the source code of different programs, and infers semantic beliefs from the database by checking four key aspects: implications of function return values, relations between function arguments, causal relationships between functions, and implicit pre- and post-conditions of functions. These four aspects are specialized to incorporate API-specific knowledge for more precise ranking and deeper semantic analysis. We describe eight such cases in APISAN that are tailored to check a variety of properties with security implications, such as cryptographic protocol API misuses, integer overflow, improper locking, and NULL dereference.

Our evaluation shows that APISAN’s approach is scal-

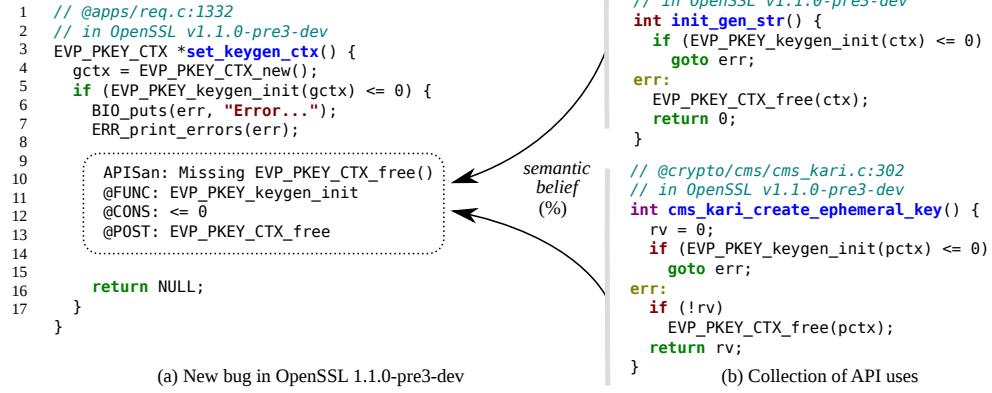


Figure 1: (a) A memory leak vulnerability found by APISAN in OpenSSL 1.1.0-pre3-dev. When a crypto key fails to initialize, the allocated context (i.e., `gctx`) should be freed. Otherwise, a memory leak will occur. APISAN first infers correct semantic usage of the API from (b) other uses of the API, and extracts a checkable rule, called a *semantic belief*, under the proper context (e.g., state: `EVP_PKEY_keygen_init() → rv <= 0 && EVP_PKEY_CTX_free()`). This newly found vulnerability has been reported and fixed in the mainstream with the patch we provided. In the above report, `@FUNC` indicates a target API, `@CONS` is a return value constraint, and `@POST` shows an expected post-action following the API.

able and effective in finding API misuses that result in critical security problems such as code execution, system hangs, or crashes. In total, we analyzed 92 million lines of code (LoC) and found 76 *previously unknown* bugs in Linux, OpenSSL, PHP, Python, and debian packages using OpenSSL (see Table 2). More importantly, we created patches for all these bugs and sent them to the mainline developers of each project. Of these, 69 bugs have been confirmed, and most have already been applied to the mainstream repositories. We are awaiting responses for the remaining reported bugs.

In short, our paper makes the following contributions:

- **New methodology.** We develop a fully automated way of finding API misuses that infers semantic beliefs from existing API uses and probabilistically ranks deviant API usages as bugs. We also formalize our approach thoroughly.
- **Practical impact.** APISAN found 76 new bugs in system software and libraries, including Linux, OpenSSL, PHP, and Python, which are 92 million LoC in total. We created patches for all bugs and most of them have already been fixed in the mainstream repositories of each project.
- **Open source tool.** We will make the APISAN framework and all its checkers publicly available online for others to readily build custom checkers on top of APISAN.

The rest of this paper is organized as follows. §2 provides an overview of APISAN. §3 describes APISAN’s design. §4 presents various checkers of APISAN. §5 describes APISAN’s implementation. §6 explains the bugs we found. §7 discusses APISAN’s limitations and potential future directions. §8 compares APISAN to previous work and §9 concludes.

2 Overview

In this section, we present an overview of APISAN, our system for finding API usage errors. These errors often have security implications, although APISAN and the principles underlying it apply to general-purpose APIs and are not limited to finding security errors in them. To find API usage errors, APISAN automatically infers semantic correctness, called *semantic beliefs*, by analyzing the source code of different uses of the API.

We motivate our approach by means of an example that illustrates an API usage error. We outline the challenges faced by existing techniques in finding the error and describe how APISAN addresses those challenges.

2.1 Running Example

Figure 1(a) shows an example of misusing the API of OpenSSL. The allocated context of a public key algorithm (`gctx` on Line 3) must be initialized for a key generation operation (`EVP_PKEY_keygen_init()` on Line 4). If the initialization fails, the allocated context should be freed by calling `EVP_PKEY_CTX_free()`. Otherwise, it results in a memory leak.

To find such errors automatically, a checker has to know the correct usage of the API. Instead of manually encoding semantic correctness, APISAN automatically infers the correct usage of an API from other uses of the API, regarding the majority usage pattern as the likely correct use. For example, considering the use of the OpenSSL API in Figure 1(a) together with other uses of the API shown in Figure 1(b), APISAN infers the majority pattern as freeing the allocated context after initialization failure (i.e., `EVP_PKEY_keygen_init() <= 0`), and thereby reports the use in Figure 1(a) as an error.

2.2 Challenges

We describe three key challenges that hinder existing approaches in finding the error in the above example.

1. Lack of specifications. A large body of work focuses on checking semantic correctness, notably dataflow analysis and model checking approaches [3, 4, 14, 17, 21, 46]. A major obstacle to these approaches is that developers should manually describe “*what is correct*,” and this effort is sometimes prohibitive in practice. To alleviate this burden, many of the above approaches check lightweight specifications, notably type-state properties [42]. These specifications are not expressive enough to capture correct API uses inferred by APISAN; for example, type-state specifications can capture finite-state rules but not rules involving a more complex state, such as the rule in the box in Figure 1(a), which states that `EVP_PKEY_CTX_free()` must be called if `EVP_PKEY_CTX_init() <= 0`. Moreover, techniques for checking such rules must track the context of the API use in order to be precise, which limits their scalability. For instance, the second example in Figure 1(b) has a constraint on `!rv`, whose tracking is necessary for precision but complicated by the presence of `goto` routines in the example.

2. Missing constraints. Engler et al. [18] find potential bugs by detecting contradictions in software in the absence of correctness semantics specified by developers. For instance, if most occurrences of a lock release operation are preceded by a lock acquire operation, then instances where the lock is released without being acquired are flagged as bugs. The premise of APISAN is similar in that the majority occurrence of an API usage pattern is regarded as likely the correct usage, and deviations are reported as bugs. However, Engler et al.’s approach does not consider semantic constraints, which can lead it to miss bugs that occur under subtle constraints, such as the one in Figure 1(a), which states that `EVP_PKEY_CTX_free()` must be called only when `EVP_PKEY_keygen_init()` fails.

3. Complex constraints. KLEE [7] symbolically executes all possible program paths to find bugs. While it is capable of tracking semantic constraints, however, it suffers from the notorious path-explosion problem; its successor, UC-KLEE [37], performs under-constrained symbolic execution that checks individual functions rather than whole programs. However, functions such as `EVP_PKEY_keygen_init()` in Figure 1 contain a function pointer, which is hard to resolve in static analysis, and cryptographic functions have extremely complex path constraints that pose scalability challenges to symbolic execution based approaches.

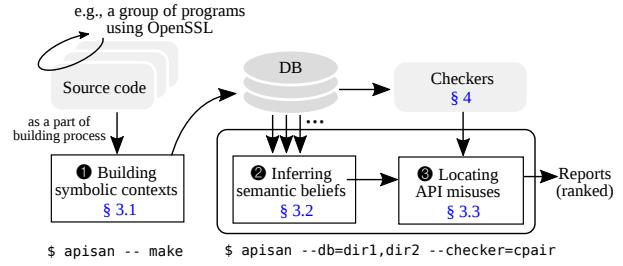


Figure 2: Overview of APISAN’s architecture and workflow. APISAN first builds symbolic contexts from existing programs’ source code and creates a database (§3.1); then APISAN infers correct usages of APIs, so-called *semantic beliefs*, in four aspects (§3.2). The inferred beliefs are used to find and rank potential API misuses to be reported as bugs (§3.3). Specific checkers are built by using the inferred beliefs and symbolic context database. If necessary, checkers incorporate domain-specific knowledge to find and rank bugs more precisely (§4).

2.3 Our Approach

APISAN’s workflow consists of three basic steps as shown in Figure 2. It first builds symbolic contexts using symbolic execution techniques on existing programs’ source code and creates a database of symbolic traces (§3.1). Then, it statistically infers correct API usages, called semantic beliefs, using the database (§3.2). Finally, it locates API misuses in the programs’ source code using the inferred beliefs and domain-specific knowledge if necessary (§3.3, §4).

We formalize our approach as a general framework, shown in Figure 5, which can be tuned using two parameters: the context checking function, which enables tailoring the checking of symbolic contexts to different API usage aspects, and an optional hint ranking function, which allows customizing the ranking of bug reports. As we will discuss shortly, our framework provides several built-in context checking functions, allowing common developers to use APISAN without modification.

Below, we describe how APISAN tackles the challenges outlined in the previous section.

1. Complete automation. In large and complex programs, it is prohibitive to rely on manual effort to check semantic correctness, such as manually provided specifications, models, or formal proofs. Instead, APISAN follows a fully automated approach, inferring semantic beliefs, i.e., correct API usages, from source code.

2. Building symbolic contexts. To precisely capture API usages involving a complex state, APISAN infers semantic beliefs from the results of symbolic execution. These results, represented in the form of symbolic constraints, on one hand contain precise semantic information about each individual use of an API, and on the other hand are abstract enough to compare across uses of the API even in different programs.

3. Relaxed symbolic execution. To prevent the path

explosion problem and achieve scalability, we perform *relaxed symbolic execution*. Unlike previous approaches, which try to explore as many paths as possible, APISAN explores as few paths as possible so as to suffice for the purpose of inferring semantic beliefs. In particular, our relaxed symbolic execution does not perform inter-procedural analysis, and unrolls loops.

4. Probabilistic ranking. To allow to prioritize developers’ inspection effort, APISAN ranks more likely bug reports proportionately higher. More specifically, APISAN’s ranking is probabilistic, denoting a confidence in each potential API misuse that is derived from a proportionate number of occurrences of the majority usage pattern, which itself is decided based on a large number of uses of the API in different programs. The ranking is easily extensible with domain-specific ranking policies for different API checkers.

3 Design of APISAN

The key insight behind our approach is that the “correctness” of API usages can be probabilistically measured from existing uses of APIs: that is, the more API patterns developers use in similar contexts, the more confidence we have about the correct API usage. APISAN automatically infers correct API usage patterns from existing source code without any human intervention (e.g., manual annotation or providing an API list), and ranks potential API misuses based on the extent to which they deviate from the observed usage pattern. To process complex, real-world software, APISAN’s underlying mechanisms for inferring, comparing, and contrasting API usages should be scalable, yet without sacrificing accuracy. In this section, we elaborate on our static analysis techniques based on relaxed symbolic execution (§3.1), methodologies to infer semantically correct API usages (§3.2), and a probabilistic method for ranking potential API misuses (§3.3).

3.1 Building Symbolic Contexts

APISAN performs symbolic execution to build symbolic contexts that capture rich semantic information for each function call. The key challenge of building symbolic contexts in large and complex programs is to overcome the path-explosion problem in symbolic execution.

We made two important design decisions for our symbolic execution to achieve scalability yet extract accurate enough information about symbolic contexts. First, APISAN localizes symbolic execution within a function boundary. Second, APISAN unrolls each loop once so that the results of symbolic execution can be efficiently represented as a *symbolic execution tree* with no backward edges. In this section, we provide justifications for

```

1 // @drivers/tty/synclink_gt.c:2363
2 // in Linux v4.5-rc4
3 static irqreturn_t slgt_interrupt(int dummy, void *dev_id) {
4     struct slgt_info *d = dev_id;
5     ...
6     for (i = 0; i < d->count; i++) {
7         if (d->ports[i] == NULL)
8             continue;
9     *     spin_lock(&d->ports[i]->lock);
10    ...
11    *     spin_unlock(&d->ports[i]->lock);
12 }
13 ...
14 return IRQ_HANDLED;
15 }
```

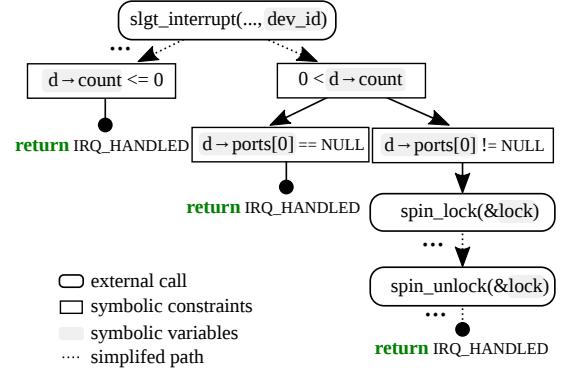


Figure 3: A typical API usage inside a loop. This code snippet comes from a `tty` device driver in the Linux v4.5-rc1. `spin_lock()` and `spin_unlock()` are used in a pair inside the loop. APISAN represents its symbolic context as a tree that contains function calls and symbolic constraints by unrolling its outer loop, as depicted at the bottom of the code snippet. Note that we use `lock` for `d->ports[0]->lock` due to space limitation.

these two design decisions within the context of finding API misuses, and provide a performance optimization that memoizes the predominant symbolic states. Finally, we precisely define the structure of symbolic execution traces computed by APISAN.

Limiting inter-procedural analysis. In APISAN, we perform symbolic execution intra-procedurally for each function. We use a fresh symbolic variable to represent each formal argument of the function, as well as the return value of each function called in its body. The symbolic constraints track C/C++ expressions over such symbolic variables, as described below. In our experience with APISAN, limiting inter-procedural analysis is reasonable for accuracy and code coverage, since most API usages can be captured within a caller function without knowing API internals.

Unrolling a loop. APISAN unrolls each loop only once to reduce the number of paths explored. While this can limit the accuracy of our symbolic execution, it does not noticeably affect the accuracy of APISAN. This is because most API usages in practice do not tend to be related to loop variables. Figure 3 (top) shows such an example in a Linux device driver. Although the symbolic context changes while executing the loop, API usages

```

(function)  $f \in \mathbb{F}$ 
(integer)  $n \in \mathbb{Z}$ , (natural)  $i \in \mathbb{N}$ 
(symbolic variable)  $\alpha ::= \langle \text{arg}, i \rangle \mid \langle \text{ret}, i \rangle$ 
(symbolic expression)  $e ::= n \mid \alpha \mid uop\ e \mid e_1\ bop\ e_2$ 
(integer range)  $r ::= [n_1, n_2]$ 
(event in trace)  $a ::= \text{call}\ f(\bar{e}) \mid \text{assume}(e, \bar{r})$ 
(trace)  $t ::= \bar{a}$ 
(database of traces)  $\mathbb{D} ::= \{ t_1, t_2, \dots \}$ 

```

Figure 4: Abstract syntax of symbolic execution traces.

of `spin_lock()` and `spin_unlock()` can be precisely captured even by unrolling the loop once. While this may not always be the case, however, we compensate for the incurred accuracy loss by collecting a larger number of API uses.

Memoizing predominant symbolic states. Another advantage of loop unrolling is that all symbolic execution traces of a function can be efficiently represented as a tree, namely, a symbolic execution tree, without having backward edges. This helps scalability because APISAN can deterministically explore the symbolic execution tree, and all intermediate results can be cached in interior nodes; most importantly, the cached results (i.e., predominant symbolic contexts) can be safely re-used because there is no control flow from a child to its ancestors. [Figure 3](#) (bottom) shows the corresponding symbolic execution tree for the function `sigt_interrupt` shown above it.

Structure of symbolic execution traces. [Figure 4](#) formally describes the structure of traces computed by APISAN using symbolic execution. Each trace t consists of a sequence of events. We refer to the i^{th} event by $t[i]$, where $1 \leq i \leq |t|$. Each event a is either a `call` to a function f with a sequence of symbolic expressions \bar{e} as arguments, or an `assume` constraint, which is a pair consisting of a symbolic expression e and its possible value ranges \bar{r} . A symbolic expression e can be a constant n , a symbolic variable α , or the result of an unary (`uop`) or binary (`bop`) operation on other symbolic expressions. Each symbolic variable α is either the return result of a function called at the i^{th} event in the trace, denoted $\langle \text{ret}, i \rangle$, or the i^{th} formal parameter of the function being symbolically executed, denoted $\langle \text{arg}, i \rangle$.

The following three traces are computed by APISAN for the code snippet in [Figure 3](#) (ignoring unseen parts)¹:

```

 $t_1 : \text{assume}(d \rightarrow \text{count}, [\text{MIN}, 0])$ 
 $t_2 : \text{assume}(d \rightarrow \text{count}, [1, \text{MAX}]);$ 
 $\quad \text{assume}(d \rightarrow \text{ports}[0], [0, 0])$ 
 $t_3 : \text{assume}(d \rightarrow \text{count}, [1, \text{MAX}]);$ 
 $\quad \text{assume}(d \rightarrow \text{ports}[0], [[\text{MIN}, -1], [1, \text{MAX}]])$ ;
 $\quad \text{call spin\_lock}(&d \rightarrow \text{ports}[0] \rightarrow \text{lock});$ 
 $\quad \text{call spin\_unlock}(&d \rightarrow \text{ports}[0] \rightarrow \text{lock})$ 

```

¹MIN and MAX stand for the minimum and maximum possible values of a related type, respectively.

3.2 Inferring Semantic Beliefs

The key challenge is to infer (most likely) correct API usages that are implicitly embedded in a large number of existing implementations. We call the inferred API usages “semantic beliefs,” not only because they are believed to be correct by a dominant number of implementations, but also because they are used in semantically similar contexts (e.g., certain state or conditions). Therefore, the more frequent the API usage patterns we observe, the stronger is the semantic belief about the correctness of API usages. APISAN infers semantic beliefs by analyzing the surrounding symbolic contexts ([§3.1](#)) without developers’ manual annotations or providing an API list.

In particular, APISAN focuses on exploring four common API context patterns.

- **Return value:** Not only does a function return the result of its computation, but it often implicates the status of the computation through the return value; for example, non-zero value in `glibc` and `PTR_ERR()` in the Linux kernel.
- **Argument:** There are semantic relations among arguments of an API; for example, the memory copy size should be smaller or equal to the buffer size.
- **Causality:** Two APIs can be causally related; for example, an acquired lock should be released at the end of critical section.
- **Conditions:** API semantics can imply certain pre- or post-conditions; for example, verifying a peer certificate is valid only if the peer certificate exists.

We give a formal description of these four patterns in [Figure 6](#) and elaborate upon them in the rest of this section. Since APISAN infers semantic beliefs, which are probabilistic in nature, there could be false positives in bug reports. APISAN addresses this problem by providing a ranking scheme for developers to check the most probable bug reports first. [Figure 5](#) formalizes this computation and [§3.3](#) presents it in further detail.

3.2.1 Implication of Return Values

Return value is usually used to return the computation result (e.g. pointer to an object) or execution status (e.g., `errno`) of a function. Especially for system programming in C, certain values are conventionally used to represent execution status. In such cases, checking the return value (execution status) properly before proceeding is critical to avoid security flaws. For instance, if a program ignores checking the return value of memory allocation (e.g., `malloc()`), it might crash later due to NULL pointer dereference. In the OpenSSL library, since the result of establishing a secure connection is passed by a return value, programs that fail to check the return value properly are vulnerable to MITM attacks [[22](#)].

$\text{SymbolicContexts}(f)$	$= \{ (t, i, C) \mid t \in \mathbb{D} \wedge i \in [1.. t] \wedge t[i] \equiv \text{call } f(*) \wedge C = \text{CONTEXTS}(t, i) \}$
$\text{Frequency}(f, c)$	$= \{ (t, i) \mid \exists C : c \in C \wedge (t, i, C) \in \text{SymbolicContexts}(f) \}$
$\text{Majority}(f)$	$= \{ c \mid \text{Frequency}(f, c) / \text{SymbolicContexts}(f) \geq \theta \}$
$\text{BugReports}(f)$	$= \{ (t, i, C) \mid (t, i, C) \in \text{SymbolicContexts}(f) \wedge C \cap \text{Majority}(f) = \emptyset \}$
$\text{BugReportScore}(f)$	$= 1 - \text{BugReports}(f) / \text{SymbolicContexts}(f) + \text{HINT}(f)$

Figure 5: The general framework of APISAN. Threshold ratio θ is used to decide whether a context c is a correct or buggy API usage. Procedures CONTEXTS and HINT are abstract; Figure 6 shows concrete instances of these procedures implemented in APISAN.

$\text{returnValueContexts} = \lambda(t, i). \{ \bar{r} \mid \exists j : t[j] \equiv \text{assume}(e, \bar{r}) \wedge \langle \text{ret}, i \rangle \in \text{retvars}(e) \}$
$\text{argRelationContexts} = \lambda(t, i). \{ (u, v) \mid t[i] \equiv \text{call } *(\bar{e}) \wedge \text{argvars}(\bar{e}[u], t) \cap \text{argvars}(\bar{e}[v], t) \neq \emptyset \}$
$\text{causalityContexts}(\bar{r}) = \lambda(t, i). \{ g \mid \exists j : t[j] \equiv \text{assume}(e, \bar{r}) \wedge \langle \text{ret}, i \rangle \in \text{retvars}(e) \wedge \exists k > j : t[k] \equiv \text{call } g(*) \}$
$\text{conditionContexts}(\bar{r}) = \lambda(t, i). \{ (g, \bar{r}') \mid \exists j : t[j] \equiv \text{assume}(e, \bar{r}) \wedge \langle \text{ret}, i \rangle \in \text{retvars}(e) \wedge \exists k > j : t[k] \equiv \text{call } g(*) \wedge \exists l : t[l] \equiv \text{assume}(e', \bar{r}') \wedge \langle \text{ret}, k \rangle \in \text{retvars}(e') \}$
$\text{defaultHint} = \lambda f. 0 \quad \text{nullDerefHint} = \lambda f. \text{if } (f\text{'s name contains } \text{alloc}) \text{ then } 0.3 \text{ else } 0$

Figure 6: Concrete instances of the CONTEXTS and HINT procedures implemented in APISAN. Function $\text{retvars}(e)$ returns all $\langle \text{ret}, i \rangle$ variables in e . Function $\text{argvars}(e, t)$ returns all $\langle \text{arg}, i \rangle$ variables in e , consulting t to recursively replace each $\langle \text{ret}, i \rangle$ variable by its associated function call symbolic expression. Both these functions are formally described in Appendix A.

Moreover, missing return value checks can lead to privilege escalation like CVE-2014-4113 [12]. Because of such critical scenarios, gcc provides a special pragma, `__attribute__((warn_unused_result))`, to enforce the checking of return values. However, it does not guarantee if a return value check is proper or not [24].

Properly checking return values seems trivial at the outset, but it is not in reality; since each API uses return values differently (e.g., `0` can be used to denote either success or failure), it is error-prone. Figure 7 shows such an example found by APISAN in Linux. In this case, `kthread_run()` returns a new `task_struct` or a non-zero error code, so the check against `0` is incorrect (Line 12).

Instead of analyzing API internals, APISAN analyzes how return values are checked in different contexts to infer proper checking of return values of an API. For an API function f , APISAN extracts all symbolic constraints on f 's return values from symbolic execution traces. After extracting all such constraints, APISAN calculates the probability of correct usage for each constraint based on occurrence count. For example, APISAN extracts how frequently the return value of `kthread_run()` is compared with `0` or `IS_ERR(p)`. APISAN reports such cases that the probability of constraints is below a certain threshold as potential bugs; the lower the probability of correctness, the more likely those cases are to be bugs.

Our framework can be easily instantiated to capture return value context by defining the context function $\text{returnValueContexts}(t, i)$, as shown in Figure 6, which extracts all checks on the return value of the function called at $t[i]$ (i.e., the i^{th} event in trace t).

3.2.2 Relations on Arguments

In many APIs, arguments are semantically inter-related. Typical examples are memory copy APIs, such as `strncpy(d, s, n)` and `memcpy(d, s, n)`; for correct operation without buffer overrun, the size of the destination

buffer d should be larger or equal to the copy length n .

APISAN uses a simple heuristic to capture possible relations between arguments. APISAN decides that two arguments are related at a function call if their symbolic expressions share a common symbolic variable. For example, the first and third arguments of `strncpy(malloc(n+1), s, n)` are considered to be related. After deciding whether a pair of arguments are related or not at each call to a function, APISAN calculates the probability of the pair of arguments being related. APISAN then classifies the calls where the probability is lower than a certain threshold as potential bugs.

Another important type of relation on arguments is the constraint on a single argument, e.g., an argument is expected to be a format string. When such constraints exist on well-known APIs like `printf()`, they can be checked by compilers. However, a compiler cannot check user-defined functions that expect a format string argument.

To capture relations on arguments, we define the context function `argRelationContexts` as shown in Figure 6. It is also straightforward to handle the format string check by extending the definition with a format check as a pair relation, such as $(-1, i)$, where -1 indicates that the pair is a special check and i denotes the i^{th} argument that is under consideration for a format check.

3.2.3 Constrained Causal Relationships

Causal relationships, also known as the a-b pattern, are common in API usage, such as `lock/unlock` and `malloc/free`. Past research [18, 29] only focuses on finding “direct” causal relationships, that is, no context constraint between two API calls. In practice, however, there are many *constrained* causal relationships as well. The conditional synchronization primitives shown in Figure 8 are one such example. In this case, there is a causal relationship between `mutex_trylock()` and `mutex_unlock()` only when `mutex_trylock()` returns a non-zero value.

```

1 // @drivers/media/usb/pvrusb2/pvrusb2-context.c:194
2 // in Linux v4.5-rc4
3 int pvr2_context_global_init(void) {
4     pvr2_context_thread_ptr = \
5         kthread_run(pvr2_context_thread_func,
6             NULL,
7             "pvrusb2-context");
8     // APISan: Incorrect return value check
9     // @FUNC: kthread_run
10    // @CONS: >= (unsigned long)-4095
11    //      < (unsigned long)-4095
12 *     return (pvr2_context_thread_ptr ? 0 : -ENOMEM);
13 }

```

Figure 7: Incorrect handling of a return value in Linux found by APISAN. `kthread_run()` returns a pointer to `task_struct` upon success or returns an error code upon failure. Because of incorrect handling of return values, this function always returns `0`, i.e., success, even in the case of error.

Both direct and constrained causality relationships can be effectively captured in the APISAN framework by defining a parametric context function $\text{causalityContexts}(\bar{r})$ shown in Figure 6, which extracts all pairs of API calls with \bar{r} as the context constraints between them. Conceptually, the parameter \bar{r} is obtained by enumerating all constraints on return values from all symbolic execution traces. In practice, however, we only check \bar{r} when necessary, for example, we only check constraints on the return value of `f()` after a call to `f()`.

3.2.4 Implicit Pre- and Post-Conditions

In many cases, there are hidden assumptions *before* or *after* calling APIs, namely, implicit pre- and post-conditions. For example, the memory allocation APIs assume that there is no integer overflow on the argument passed as allocation size, which implies that there should be a proper check before the call. Similarly, `SSL_get_verify_result()`, an OpenSSL API which verifies the certificate presented by the peer, is meaningful only when `SSL_get_peer_certificate()` returns a non-NULL certificate of a peer, though which could happen either before or after `SSL_get_verify_result()`. So the validity check of a peer certificate returned by `SSL_get_peer_certificate()` is an implicit pre- or post-condition of `SSL_get_verify_result()`.

Similar to the context checking of causal relationships, we define a parametric context function $\text{conditionContexts}(\bar{r})$ shown in Figure 6, to capture implicit pre- and post-conditions of an API call. Here, the parameter \bar{r} serves as the pre-condition, and the post-condition is extracted along with the called API.

3.3 Ranking Semantic Disbeliefs

After collecting the API usage patterns discussed above, APISAN statistically infers the majority usage patterns for each API function under each context. This computation is described in detail in Figure 5. Intuitively,

```

1 // @kernel/workqueue.c:1977
2 // in Linux v4.5-rc4
3 static bool manage_workers(struct worker *worker)
4 {
5     struct worker_pool *pool = worker->pool;
6     if (!mutex_trylock(&pool->manager_arb))
7         return false;
8     pool->manager = worker;
9     maybe_create_worker(pool);
10    pool->manager = NULL;
11    mutex_unlock(&pool->manager_arb);
12    return true;
13 }

```

Figure 8: An example usage of conditional locking in Linux. `mutex_trylock()` returns non-zero value when a lock is acquired. So `mutex_unlock()` is necessary only in this case.

APISAN labels an API usage pattern as majority (i.e., likely correct usage) if its occurrence ratio is larger than a threshold θ . In our experience, this simple approach is quite effective, though more sophisticated statistical approaches could be further applied. Each call to a function that deviates from its majority usage pattern is reported as a potential bug.

Since our approach is probabilistic in nature, a bug report found by APISAN might be a false alarm. APISAN ranks bug reports in decreasing order of their likelihood of being bugs, so that the most likely bugs have the highest priority to be investigated. Based on the observation that the more the majority patterns repeat, the more confident we are that these majority patterns are correct specifications, APISAN uses the ratio of majority patterns over “buggy” patterns as a measure of the likelihood. In addition, APISAN can also adjust the ranking with domain-specific knowledge about APIs. For example, if an API name contains a sub-string `alloc`, which indicates that it is very likely to handle memory allocation, we can customize APISAN to give more weight for its misuse in the return value checking.

4 Checking API Misuses

In this section, we demonstrate how inferred semantic beliefs described in the previous section can be used to find API misuses. In particular, we introduce eight cases, which use API-specific knowledge for more precise ranking and deeper semantic analysis.

4.1 Checking SSL/TLS APIs

A recent study shows that SSL/TLS APIs are very error-prone—especially, validating SSL certificates is “*the most dangerous code in the world*” [22]. To detect their incorrect use, specialized checkers that rely on hand-coded semantic correctness have been proposed [22, 26].

In APISAN, we easily created a SSL/TLS checker based on the constraints of return values and implicit pre- and post-conditions without manually coding seman-

```

1 // @librabbitmq/amqp_openssl.c:180
2 // in librabbitmq v0.8
3 static int
4 amqp_ssl_socket_open(void *base, const char *host,
5                      int port, struct timeval *timeout) {
6     // APISan: Missing implicit condition
7     // @FUNC : SSL_get_verify_result
8     // @CONS : == X509_V_OK
9     // @COND : SSL_get_peer_certificate != NULL
10    cert = SSL_get_peer_certificate(self->ssl);
11    result = SSL_get_verify_result(self->ssl);
12    if (X509_V_OK != result) {
13        if (!cert || X509_V_OK != result) {
14            goto error_out3;
15        }
16    }

```

Figure 9: Incorrect use of OpenSSL API found in librabbitmq, a message queuing protocol library, by APISAN. `SSL_get_verify_result()` always returns `X509_V_OK` if there is no certificate (i.e., `!cert`). So `SSL_get_peer_certificate()` needs to be validated before or after calling `SSL_get_verify_result()`.

tic correctness. In practice, as we described in §3.2.4, the sequence of API calls and relevant constraints to validate SSL certificates can be captured by using implicit pre- and post-conditions. For example, Figure 9 shows that APISAN successfully inferred valid usage of `SSL_get_verify_result()` and discovered a bug.

4.2 Checking Integer Overflow

Integer overflows remain a very important threat despite extensive research efforts for checking them. Checkers have to deal with two problems: (1) whether there is a potential integer overflow, and (2) whether such a potential integer overflow is exploitable. KINT [45], the state-of-the-art integer security checker, relies on scalable static analysis to find potential integer overflows. To decide exploitability, KINT relies on users’ annotations on untrusted data source and performs taint analysis to decide whether untrusted sources are related to an integer overflow. But if annotations are missing, KINT may miss some bugs.

Instead of annotating untrusted sources, APISAN infers *untrusted sinks* to decide that an integer overflow has security implications. The background belief is “checking sinks implies that such sinks are untrusted.” APISAN considers APIs with arguments that are untrusted sinks as *integer overflow-sensitive APIs*. To infer whether an API is integer overflow-sensitive, the checker extracts all function calls whose arguments have arithmetic operations that can result in integer overflow. The checker classifies such function calls into three categories: (1) correct check, (2) incorrect check, and (3) missing check. If an argument has a constraint that prevents integer overflow, then it is a correct check. Determining potential integer overflow is straightforward because APISAN maintains a numerical range for each symbolic variable. If such a constraint cannot prevent integer overflow, then it is an

```

1 // @fs/ext4/resize.c:193
2 // in Linux v4.5-rc4
3 static struct ext4_new_group_data
4     *alloc_flex_gd(unsigned long flexbg_size)
5 {
6     if (flexbg_size >=
7         UINT_MAX / sizeof(struct ext4_new_group_data))
8         goto out2;
9     flex_gd->count = flexbg_size;
10    // APISan: Incorrect integer overflow check
11    // @CONS: flexbg_size < UINT_MAX / 20
12    // @EXPR: flexbg_size * 40
13    flex_gd->groups =
14        kmalloc(sizeof(struct ext4_new_group_data) *
15                flexbg_size, GFP_NOFS);
16 }

```

Figure 10: An integer overflow vulnerability found in Linux by APISAN. Since `struct ext4_new_group_data` is larger than `struct ext4_new_flex_group_data`, previous overflow check can be bypassed. Interestingly, this bug was previously found by KINT and already patched [8], but APISAN found the patch is actually incorrect.

incorrect check. Finally, if there is no constraint, then it is a missing check. The checker concludes that an API is more integer overflow-sensitive if the ratio of correct checks over total checks is higher. The checker gives a higher rank to incorrect checks followed by missing checks. For example, Figure 10 shows an integer overflow vulnerability found by APISAN.

4.3 Checking Memory Leak

A memory leak can be represented as a causal relationship between memory allocation and free functions. As Figure 1 shows, APISAN can infer a constrained causal relation between such a pair of functions, which may not be captured as a direct causal relation. When a function that is presumed to be a free function is not called following a function that is presumed to be the corresponding allocation function, it is reported as a memory leak with a higher rank. In this manner, APISAN effectively captures typical usage patterns of memory allocation and free routines to report potential memory leaks.

4.4 Checking Lock and Unlock

Similar to checking memory leaks, lock checking is based on a constrained causal relationship between lock and unlock functions inferred by APISAN. It gives a higher rank to cases where there are missing unlock function calls in some of the paths. For example, Figure 11 shows that there is one missing `clk_prepare_unlock()` call among two symbolic execution paths.

4.5 Checking NULL Dereference

NULL dereference can happen by accessing a pointer returned by a memory allocation function, such as `malloc()` and `kmalloc()`, without validation. Checking NULL

```

1 // @drivers/clk/clk.c:2672
2 // in Linux v4.5-rc4
3 void clk_unregister(struct clk *clk) {
4     clk_prepare_lock();
5     if (clk->core->ops == &clk_nodrv_ops) {
6         pr_err("%s: unregistered clock: %s\n", __func__,
7                clk->core->name);
8         // APISan: Missing clk_prepare_unlock()
9         // @FUNC: clk_prepare_lock
10        // @CONS: None
11        // @POST: clk_prepare_unlock
12        return;
13    }
14    clk_prepare_unlock();
15 }

```

Figure 11: A missing unlock bug in Linux found by APISAN. It shows a common pattern of violating a causal relation.

dereference is based on the return value inference of APISAN. It collects how frequently the return value of a function is compared against NULL. Based on this information, it can find missing NULL checks. In addition, it gives a higher rank to cases where the function name contains common keywords for allocation such as alloc or new.

4.6 Checking Return Value Validation

Checking a return value of a function properly is more important than checking a return value itself. If the return value is incorrectly checked, the caller is likely to believe that the callee succeeded. Moreover, it is quite usual that incorrect checks fail only in rare cases, so that finding such incorrect checks is much more difficult than completely omitted checks. APISAN can find bugs of this kind, such as the one shown in Figure 7, by comparing constraints of return value checks.

4.7 Checking Broken Argument Relation

We can find potential bugs by inferring and finding broken relations between arguments. However, detecting a broken relation does not mean that it is always a bug, because there might be an implicit relation between two arguments that cannot be captured by APISAN (e.g., complex pointer aliasing of the buffer). This lack of information is complemented by a ranking policy that incorporates domain-specific knowledge, for example, a broken argument relation is ranked higher if either argument has a sizeof() operator.

4.8 Checking Format String

Incorrect use of format strings is one frequent source of security vulnerabilities [39]. Modern compilers (e.g., gcc) give compile-time warnings for well-known APIs such as printf(). However, in the case of programs that have their own printf-like functions (e.g., PHP), compilers cannot detect such errors.

To infer whether a function argument is a format string, we use a simple heuristic: if the majority of symbolic expressions for an argument is a constant string and contains well-known format codes (e.g, %s), then the argument is considered as a format string. For the cases where a symbolic variable is used as a format string argument, the corresponding API calls will be considered as potential bugs. Similarly, domain-specific knowledge can be applied as well. Bug reports of an API whose name contains a sub-string print is ranked higher, since it indicates that the API is very likely to take a format string as an argument.

5 Implementation

APISAN is implemented in 9K lines of code (LoC) as shown in Table 1: 6K of C/C++ for generating symbolic execution traces, which is based on Clang 3.6, and 3K of Python for checkers and libraries. We empirically chose a threshold value of **0.8** for deciding whether to label an API usage pattern as majority. Since APISAN ranks all reports in order of bug likelihood, however, the result is not sensitive to the threshold value in that the ordering of the top-ranked reports remains the same.

Component	Lines of code
Symbolic database generator	6,256 lines of C/C++
APISAN Library	1,677 lines of Python
Checkers	1,047 lines of Python
Total	8,980 lines of code

Table 1: Components and lines of code of APISAN.

6 Evaluation

To evaluate APISAN, this section attempts to answer the following questions:

- How effective is APISAN in finding previously unknown API misuses? (§6.1)
- How easy is APISAN to use by end-users and checker developers? (§6.2)
- How reasonable is APISAN’s relaxed symbolic execution in finding bugs? (§6.3)
- How effective is APISAN’s approach in ranking bugs? (§6.4)
- How effective is APISAN’s approach compared to manual checking? (§6.5)

6.1 New Bugs

We applied APISAN to Linux v4.5-rc4, OpenSSL 1.1.0-pre3-dev, PHP 7.0, Python 3.6, and all 1,204 debian packages using the OpenSSL library. APISAN generated 40,006 reports in total, and we analyzed the reports

Program	Module	API misuse	Impact	Checker	#bugs	S.
Linux	cifs/cifs_dfs_ref.c	heap overflow	code execution	args	1	✓
	xenbus/xenbus_dev_frontend.c	missing integer overflow check	code execution	intovfl	1	✓
	ext4/resize.c	incorrect integer overflow check	code execution	intovfl	1	✓
	tipc/link.c	missing tipc_bccast_unlock()	deadlock	cpair	1	✓
	clk/clk.c	missing clk_prepare_unlock()	deadlock	cpair	1	✓
	hotplug/acpiphp_glue.c	missing pci_unlock_rescan_remove()	deadlock	cpair	1	✓
	usbvision/usbvision-video.c	missing mutex_unlock()	deadlock	cpair	1	✓
	drm/drm_dp_mst_topology.c	missing drm_dp_put_port()	DoS	cpair	1	✓
	affs/file.c	missing kunmap()	DoS	cpair	1	✓
	acpi/sysfs.c	missing kobject_create_and_add() check	system crash	rvchk	1	✓
	cx231xx/cx231xx-417.c	missing kmalloc() check	system crash	rvchk	1	✓
	qxl/qxl_kms.c	missing kmalloc() check	system crash	rvchk	1	P
	chips/cfi_cmdset_0001.c	missing kmalloc() check	system crash	rvchk	1	✓
	ata/sata_sx4.c	missing kzalloc() check	system crash	rvchk	1	✓
	hsi/hsi.c	missing kzalloc() check	system crash	rvchk	2	✓
	mwiflex/sdio.c	missing kzalloc() check	system crash	rvchk	2	✓
	usbtv/usbtv-video.c	missing kzalloc() check	system crash	rvchk	1	✓
	cxgb4/clip_tbl.c	missing t4_alloc_mem() check	system crash	rvchk	1	✓
	devfreq/devfreq.c	missing devm_kzalloc() check	system crash	rvchk	2	✓
	i915/intel_dsi_panel_vbt.c	missing devm_kzalloc() check	system crash	rvchk	1	✓
	gpio/gpio-mcp23s08.c	missing devm_kzalloc() check	system crash	rvchk	1	✓
	drm/drm_crtc.c	missing drm_property_create_range() check	system crash	rvchk	13	✓
	gma500/framebuffer.c	missing drm_property_create_range() check	system crash	rvchk	1	✓
	emu10k1/emu10k1_main.c	missing kthread_create() check	system crash	rvchk	1	✓
	m5602/m5602_s5k83a.c	missing kthread_create() check	system crash	rvchk	1	✓
	hisax/isdnl2.c	missing skb_clone() check	system crash	rvchk	1	✓
	qlcnic/qlcnic_ctx.c	missing qlcnic_alloc_mb_args() check	system crash	rvchk	1	✓
	xen-netback/xenbus.c	missing vzalloc() check	system crash	rvchk	1	✓
	i2c/ch7006_drv.c	missing drm_property_create_range() check	system crash	rvchk	1	✓
	fmc/fmc-fakedev.c	missing kmemdup() check	system crash	rvchk	1	P
	rc/igorplugusb.c	missing rc_allocate_device() check	system crash	rvchk	1	✓
	s5p-mfc/s5p_mfc.c	missing create_singlethread_workqueue() check	system crash	rvchk	1	P
	fusion/mptbase.c	missing create_singlethread_workqueue() check	system crash	rvchk	1	P
	nes/nes_cm.c	missing create_singlethread_workqueue() check	system crash	rvchk	1	✓
	dvb-usb-v2/mxl111sf.c	missing mxl111sf_enable_usb_output() check	malfunction	rvchk	2	✓
	misc/xen-kbdfront.c	missing xenbus_printf() check	malfunction	rvchk	1	✓
	pvrusb2/pvrusb2-context.c	incorrect kthread_run() check	malfunction	rvchk	1	P
	agere/et131x.c	incorrect drm_alloc_coherent() check	malfunction	rvchk	1	✓
	drbd/drbd_receiver.c	incorrect crypto_alloc_hash() check	malfunction	rvchk	1	✓
	mlx4/mlx4.c	incorrect mlx4_alloc_cmd_mailbox() check	maintanence	rvchk	1	✓
	usnic/usnic_ib_qp_grp.c	incorrect kzalloc() check	maintanence	rvchk	2	✓
	aoe/aoecmd.c	incorrect kthread_run() check	maintanence	rvchk	1	✓
	ipv4/tcp.c	incorrect crypto_alloc_hash() check	maintanence	rvchk	1	✓
	mfd/bcm590xx.c	incorrect i2c_new_dummy() check	maintanence	rvchk	1	P
	usnic/usnic_ib_main.c	incorrect ib_alloc_device() check	maintanence	rvchk	1	✓
	usnic/usnic_ib_qp_grp.c	incorrect usnic_fwd_dev_alloc() check	maintanence	rvchk	1	✓
OpenSSL	dsa/dsa_gen.c	missing BN_CTX_end()	DoS	cpair	1	✓
	apps/req.c	missing EVP_PKEY_CTX_free()	DoS	cpair	1	✓
	dh/dh_pmeth.c	missing OPENSSL_memdup() check	system crash	rvchk	1	✓
PHP	standard/string.c	missing integer overflow check	code execution	intovfl	3	✓
	phpdbg/phpdbg_prompt.c	format string bug	code execution	args	1	✓
Python	Modules/zipimport.c	missing integer overflow check	code execution	intovfl	1	✓
rabbitmq	librabbitmq/amqp_openssl.c	incorrect SSL_get_verify_result() use	MITM	cond	1	✓
hexchat	common/server.c	incorrect SSL_get_verify_result() use	MITM	cond	1	✓
lprng	auth/ssl_auth.c	incorrect SSL_get_verify_result() use	MITM	cond	1	P
afflib	lib/afftest.cpp	missing BIO_new_file() check	system crash	rvchk	1	✓
	tools/aff_bom.cpp	missing BIO_new_file() check	system crash	rvchk	1	✓

Table 2: List of new bugs discovered by APISAN. We sent patches of all 76 new bugs; 69 bugs have been already confirmed and applied by corresponding developers (marked ✓ in the rightmost column); 7 bugs (marked P in the rightmost column) have not been confirmed yet. APISAN analyzed 92 million LoC and found one bug per 1.2 million LoC.

according to ranks. As a result, APISAN found 76 previously unknown bugs: 64 in Linux, 3 in OpenSSL, 4 in PHP, 1 in Python, and 5 in the debian packages (see Table 2 for details). We created patches for all the bugs and sent them to the mainline developers of each project. 69 bugs have been confirmed by the developers and most have already been applied to the mainline repositories. For remaining, 7 bugs, we are waiting for their response.

Security implications. All of the bugs we found have serious security implications: e.g., code execution, system crash, MITM, etc. For a few bugs including integer overflows in Python(CVE-2016-5636 [13]) and PHP, we could even successfully exploit them by chaining ROP gadgets [2, 27]. In addition, we found that the vulnerable Python module is in the whitelist of Google App Engine and reported it to Google.

6.2 Usability

End-users. APISAN can be seamlessly integrated into an existing build process. Users can generate symbolic execution databases by simply invoking the existing build command, e.g., `make`, with apisan.

```
1 # generate DB
2 $ apisan make
```

With the database, users can run various checkers, which extract semantic beliefs from the database and locate potential bugs in order of their likelihood. For eight types of API misuses described at §4, we developed five checkers: return value checker (`rvchk`), causality checker (`cpair`), argument relation checker (`args`) implicit pre- and post-condition checker (`cond`), and integer overflow checker (`intovfl`).

```
1 # run a causality checker
2 $ apisan --checker=cpair
3 @FUNC: EVP_PKEY_keygen_init
4 @CONS: ((-2147483648, 0),)
5 @POST: EVP_PKEY_CTX_free
6 @CODE: { req.c:1745 }
7 ...
```

APISAN can also be run against multiple databases generated by different project code repositories. For example, users can infer semantic beliefs from multiple programs (e.g., all packages using libssl) and similarly get a list of ranked, potential bugs. This is especially useful for relatively young projects, which lack sufficient API usages.

```
1 # check libssl misuses by using rabbitmq and hexchat repos
2 $ apisan --checker=cond --db=rabbitmq,hexchat
```

Checker developers. Developing specialized checkers is easy; APISAN provides a simple interface to access symbolic execution databases. Each of our checkers is around 200 lines of Python code as shown in §5. Providing API-specific knowledge such as manual annotations can be easily integrated in the Python script.

		UC-KLEE	APISAN
Approach	Loop Inter-procedural Constraint	best effort yes SAT	once no numerical range
Bugs (OpenSSL)	Memory leak NULL dereference Uninitialized data	5 - 6	7 (2*) 11 -

Table 3: Comparison between UC-KLEE and APISAN in approaches and bugs found in OpenSSL v1.0.2, which is used in UC-KLEE’s evaluation [37]. APISAN found 7 memory leak bugs and 11 NULL dereference vulnerabilities; two memory leak bugs (marked *) were previously unknown, and our two patches have been applied to the mainline repository.

6.3 Effect of Relaxed Symbolic Execution

One of our key design decisions is to use relaxed symbolic execution for scalability at the cost of accuracy. To evaluate the effect of this design decision, we compare APISAN against UC-KLEE, which performs best-effort accurate symbolic execution including inter-procedural analysis and best-effort loop unrolling. For comparison, we ran UC-KLEE and APISAN on OpenSSL v1.0.2, which is the version used for UC-KLEE’s evaluation. Table 3 shows a summary of the result.

APISAN found 11 NULL dereference bugs caused by missing return value checks of `OPENSSL_malloc()`, which are already fixed in the latest OpenSSL. Also, APISAN found seven memory leak bugs related to various APIs, such as `BN_CTX_new()`, `BN_CTX_start()`, and `EVP_PKEY_CTX_new()`, without any annotations. Two of these bugs were previously unknown; we sent patches which were confirmed and applied to the OpenSSL mainline. UC-KLEE found five memory leak bugs related to `OPENSSL_malloc()` with the help of users’ annotations.

Interestingly, there is no common bug between UC-KLEE and APISAN. UC-KLEE cannot find the bugs that APISAN found because of function pointers, which are frequently used for polymorphism, and path explosion in complex cryptographic operations. APISAN does not discover the five memory bugs that UC-KLEE found because of diverse usages of `OpenSSL_malloc()`. Also, APISAN could not find any uninitialized memory bugs since it does not track memory accesses.

6.4 Ranking Effectiveness

Another key design aspect of APISAN is its ranking scheme. In this section, we investigate two aspects of our ranking scheme: (1) where true-positives are located in bug reports and (2) what are typical reasons of false positives. To this end, we analyzed the results of the return value checker (`rvchk`) on Linux v4.5-rc4.

True positives. If true-positive reports are highly ranked, developers can save effort in investigating bug reports. An

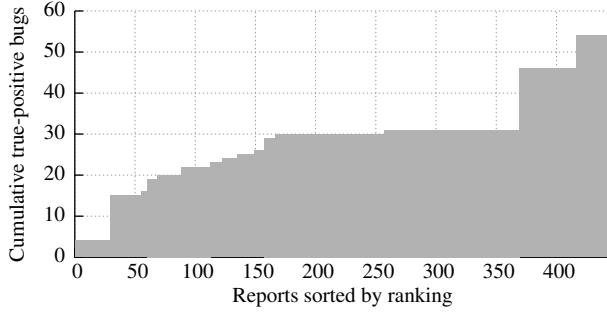


Figure 12: Cumulative true-positive bugs in Linux v4.5-rc4 reported by our return value checker (rvchk). We investigated top 445 bug reports out of 2,876 reports in total. Most new bugs are highly ranked.

author audited the top 445 reports out of 2,876 reports for two days and found 54 new bugs. As shown in Figure 12, most new bugs are highly ranked. This shows that our ranking scheme is effective to save developers’ effort by letting them investigate the highest-ranked reports first.

False positives. To understand what causes false positives, we manually investigated all false positive cases in the top 445 reports, and found a few frequent reasons: diverse patterns of return value checking, wrapper functions delegating return value checking to callers, and semantically correct, but rare patterns.

Some kernel APIs, such as `snd_pcm_new()` [40], return zero on success or a negative error code on failure. In this case, there are two valid ways to check for error: comparison against zero (i.e., $\text{== } 0$) or negative value (i.e., < 0). If the majority of code follows one pattern (`snd_pcm_new() < 0`), APISAN flags the minor correct cases as bugs.

Some wrapper functions delegate return value checking to their callers. APISAN treats these cases as if return value checking is missing because APISAN does not perform inter-procedural analysis.

If a return value of a function can have multiple meanings, APISAN can decide the rare cases as bugs. For example, most functions use `strcmp()` to test if two strings are equivalent (i.e., $\text{== } 0$). But for the rare cases, which in fact use `strcmp()` to decide alphabetical order of two strings (i.e., < 0), APISAN generates false alarms.

6.5 Comparison with Manual Auditing

The other extreme to automatic bug finding is manual auditing by developers. Manual auditing would be the most accurate but is not scalable in size and cost. We compared APISAN with manual auditing to grasp how accurate APISAN is compared to the ground truth.

To this end, we manually inspected memory allocation and free functions in OpenSSL v1.1.0-pre3-dev because OpenSSL faithfully follows naming conventions: allocation functions end with `_new` or `alloc`, and free functions end with `_free`.

```

1 // @ext/standard/string.c:877
2 // in PHP v5.5.9-rc1
3 PHP_FUNCTION(wordwrap) {
4     if (linelength > 0) {
5         chk = (int)(textlen/linelength + 1);
6         // no integer overflow
7         newtext = safe_emalloc(chk, \
8             breakcharlen, textlen + 1);
9         allocated = textlen + chk * breakcharlen + 1;
10    }
11 }

```

```

1 // @ext/standard/string.c:946
2 // in PHP v7.0.0-rc1
3 PHP_FUNCTION(wordwrap) {
4     if (linelength > 0) {
5         chk = (size_t)(ZSTR_LEN(text)/linelength + 1);
6         // introduce a new integer overflow
7         newtext = zend_string_alloc( \
8             chk * breakchar_len + ZSTR_LEN(text), 0);
9         allocated = ZSTR_LEN(text) + chk * breakchar_len + 1;
10    }
11 }

```

Figure 13: An integer overflow bug introduced by changing string allocation API in PHP. While the old string allocation API, `safe_emalloc()`, internally checks integer overflow, the new API, `zend_string_alloc()` has no such check.

To determine how APISAN accurately infers the correct check of return value, we counted how many allocation functions are inferred to need NULL checking by APISAN. Among 294 allocation functions, APISAN successfully figured out that 164 allocation functions require NULL checking. To assess the accuracy of APISAN’s causal relation inference, we counted how many allocation-free functions are inferred as causal relations by APISAN. APISAN found 37 pairs out of 187 such causal relations.

The inaccuracy of APISAN mainly stems from a small number of API usages and limited symbolic execution. For example, if allocated memory is freed by a callback function, APISAN fails to detect the causal relation.

6.6 Performance

Our experiments are conducted on a 32-core Xeon server with 256GB RAM. Constructing a symbolic database for Linux kernel, a one-time task for analysis, takes roughly eight hours and generates 300 GB database. Each checker takes approximately six hours. Thus, APISAN can analyze a large system in a reasonable time bound.

6.7 Our Experience with APISAN

While investigating the bug reports generated by APISAN, we found several interesting bugs, which were introduced while fixing bugs or refactoring code to reduce potential bugs. We believe that it shows that bug fixing is the essential activity during the entire life cycle of any software, and automatic bug finding tools such as APISAN should be scalable enough for them to be integrated into the daily software development process.

Incorrect bug fixes. Interestingly, APISAN found an incorrect bug patch, which was found and patched by KINT [45]. The bug was a missing integer overflow check in ext4 file system, but the added condition was incorrect [8]. Also, the incorrect patch was present for almost four years, showing the difficulty of finding such bugs that can be reproduced only under subtle conditions. Since APISAN gives a higher rank for incorrect condition check for integer overflow, we easily found this bug.

Incorrect refactoring. While investigating PHP integer overflow bugs in Figure 13, we found that the bug was newly introduced when changing string allocation APIs; the new string allocation API, `zend_string_alloc()`, omits an internal integer overflow check, making its callers vulnerable to integer overflow.

7 Discussion

In this section, we discuss the limitations of APISAN’s approach and discuss potential future directions to mitigate the limitations.

Limitations. APISAN does not aim to be sound nor complete. In fact, APISAN has false positives (§6.4) as well as false negatives (§6.3, §6.5).

Replacing manual annotations. One practical way to reduce false negatives is to run multiple checkers on the same source code. In this case, APISAN’s inference results can be used to provide missing manual annotations required by other checkers. For example, APISAN can provide inferred integer overflow-sensitive APIs to KINT and inferred memory allocation APIs to UC-KLEE.

Interactive ranking and filtering. In our experience, the false positive reports of APISAN are repetitive since incorrect inference of an API can incur many false positive reports. Therefore, we expect that incorporating the human feedback of investigation into APISAN’s inference and ranking will significantly reduce false positives and developers’ investigation efforts.

Self regression. As we showed in §6.7, bug fixing and refactoring can introduce new bugs. APISAN’s approach is also a good fit for self-regression testing by comparing two versions of bug reports and giving higher priorities to changed results.

8 Related Work

In this section, we survey related work in bug finding, API checking, and semantic inference.

Finding bugs. Meta compilation [3, 17, 25] performs static analysis integrated with compilers to enforce domain-specific rules. RacerX [16] proposed flow-sensitive static analysis for finding deadlocks and race

conditions. LCLint [20] detects mismatches between source code and user-provided specifications. Sparse [41] is a static analysis tool to find certain types of bugs (e.g., mixing pointers to user and kernel address spaces, and incorrect lock/unlock) in the Linux kernel based on developers’ annotations. Model checking has been applied to various domains including file systems [24, 38, 48, 49], device drivers [5], and network protocols [34]. A frequent obstacle in applying these techniques is the need to specify semantic correctness, e.g., domain-specific rules and models. In contrast, APISAN statistically infers semantic correctness from source code; it is generic without requiring models or annotations, but it could incur higher false positives than techniques that use precise semantic correctness information.

Checking API usages. SSLint [26] is a static analysis tool to find misuses of SSL/TLS APIs based on predefined rules. MOPS [9] checks source code against security properties, i.e., rules of safe programming practices. Jfern [46] models common vulnerabilities into graph traversals in a code property graph. Unlike these solutions, which are highly specialized for a certain domain (or an API set) and rely on hand-coded rules, APISAN is generally applicable to any domain without manual effort.

Inferring semantics. Engler et al. [18] find deviations from the results of static analysis. Juxta [32] finds deviations by comparing multiple file systems, which follow similar specifications. APISAN’s goal is to find deviations in API usages under rich symbolic contexts. DynaMine [30] and VCCFinder [36] automatically extract bug patterns from source code repositories by analyzing bug patches. These approaches would be useful in APISAN as well.

Automatic generation of specifications has been explored by Kremenek et al. [28] for resource allocation, by PRMiner [29] for causal relations, by APIMiner [1] for partial ordering of APIs, by Daikon [19] from dynamic execution traces, by Taghdiri et al. [43] for structural properties, by PRIME [33] for temporal specifications, by Nguyen et al. [35] for preconditions of APIs, by Gruska et al. [23] for sequences of functions, by JIGSAW [44] for resource accesses, by MERLIN [31] for information flow specifications, and by Yamaguchi et al. [47] for taint-style vulnerabilities. These approaches focus on extracting one aspect of the specification. Also, some of them [1, 43] are not scalable because of the complexity of the algorithms used. On the other hand, APISAN focuses on extracting four orthogonal aspects of API usages and using them in combination to find complex bug patterns.

9 Conclusion

We proposed APISAN, a fully automated system for finding API usage bugs by inferring and contrasting semantic beliefs about API usage from source code. We applied APISAN to large, widely-used software, including the Linux kernel, OpenSSL, PHP, and Python, composed of 92 million lines of code. We found 76 previously unknown bugs of which 69 bugs have already been confirmed. Our results show that APISAN’s approach is effective in finding new bugs and is general enough to extend easily to custom API checkers based on APISAN.

10 Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This work was supported by DARPA under agreement #15-15-TC-FP-006, #HR0011-16-C-0059 and #FA8750-15-2-0009, NSF awards #CNS-1563848, #DGE-1500084, #1253867 and #1526270, ONR N00014-15-1-2162, ETRI MSIP/IITP[B0101-15-0644], and NRF BSRP/MOE[2015R1A6A3A03019983]. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon.

References

- [1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the 6th joint meeting of European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, Dubrovnik, Croatia, Sept. 2007.
- [2] An integer overflow bug in `php_str_to_str_ex()` led arbitrary code execution. <https://bugs.php.net/bug.php?id=71450>, 2016.
- [3] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (Oakland)*, pages 143–160, Oakland, CA, May 2002.
- [4] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.
- [5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the ACM EuroSys Conference*, pages 73–85, Leuven, Belgium, Apr. 2006.
- [6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, Alexandria, VA, Oct.–Nov. 2006.
- [7] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.
- [8] H. Chen. [PATCH] FS: ext4: fix integer overflow in `alloc_flex_gd()`. <http://lists.openwall.net/linux-ext4/2012/02/20/42>, 2012.
- [9] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, Nov. 2002.
- [10] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2002.
- [11] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.
- [12] CVE-2014-4113. <http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-4113>, 2014.
- [13] CVE-2016-5636. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5636>, 2016.
- [14] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI’02*, 2002.
- [15] M. S. Dittmer and M. V. Tripunitara. The UNIX process identity crisis: A standards-driven approach to setuid. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, Scottsdale, Arizona, Nov. 2014.
- [16] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.
- [17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Oct. 2000.
- [18] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.
- [19] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, Los Angeles, CA, USA, May 1999.
- [20] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the 1994 ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, New Orleans, Louisiana, USA, Dec. 1994.
- [21] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17(2), 2008.
- [22] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, Raleigh, North Carolina, Oct. 2012.
- [23] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 2010 International Symposium on Software Testing and Analysis (ISSTA)*, Trento, Italy, July 2010.
- [24] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *Proceedings of the 6th Usenix Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, Feb. 2008.
- [25] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of*

- the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
- [26] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang. Vetting SSL usage in applications with SSLint. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.
- [27] Heap overflow in zipimporter module. <https://bugs.python.org/issue26171>, 2016.
- [28] T. Kremeneck, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.
- [29] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference (ESEC) held jointly with 13th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, Lisbon, Portugal, Sept. 2005.
- [30] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference (ESEC) held jointly with 13th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, Lisbon, Portugal, Sept. 2005.
- [31] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification Inference for Explicit Information Flow Problems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009.
- [32] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.
- [33] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the 2012 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Tucson, AZ, USA, Oct. 2012.
- [34] M. S. Musuvathi, D. Park, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.
- [35] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining pre-conditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, Hong Kong, Sept. 2014.
- [36] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Denver, Colorado, Oct. 2015.
- [37] D. A. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *Proceedings of the 24th Usenix Security Symposium (Security)*, Washington, DC, Aug. 2015.
- [38] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpacı-Dusseau, and A. C. Arpacı-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–280, Dublin, Ireland, June 2009.
- [39] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.
- [40] snd_pcm_new(). <https://www.kernel.org/doc/html/docs/device-drivers/API-snd-pcm-new.html>, 2016.
- [41] Sparse - a Semantic Parser for C. https://sparse.wiki.kernel.org/index.php/Main_Page, 2013.
- [42] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1), 1986.
- [43] M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. In *Proceedings of the 19th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Linz, Austria, Sept. 2004.
- [44] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger. JIGSAW: Protecting resource access by inferring programmer expectations. In *Proceedings of the 23rd Usenix Security Symposium (Security)*, San Diego, CA, Aug. 2014.
- [45] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.
- [46] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.
- [47] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.
- [48] J. Yang, P. Twohey, and Dawson. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–288, San Francisco, CA, Dec. 2004.
- [49] J. Yang, C. Sar, and D. Engler. eXplode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 10–10, Seattle, WA, Nov. 2006.

A Appendix

Function $\text{retvars}(e)$ returns all $\langle \text{ret}, i \rangle$ variables in e , which is defined as follows:

$$\text{retvars}(e) = \begin{cases} \emptyset & \text{if } e \equiv n \\ \emptyset & \text{if } e \equiv \langle \text{arg}, i \rangle \\ \{\langle \text{ret}, i \rangle\} & \text{if } e \equiv \langle \text{ret}, i \rangle \\ \text{retvars}(e') & \text{if } e \equiv uop e' \\ \text{retvars}(e_1) \cup \text{retvars}(e_2) & \text{if } e \equiv e_1 \text{ bop } e_2 \end{cases}$$

Function $\text{argvars}(e, t)$ returns all $\langle \text{arg}, i \rangle$ variables in e , consulting t to recursively replace each $\langle \text{ret}, i \rangle$ variable by its associated function call symbolic expression. It is defined as follows:

$$\text{argvars}(e, t) = \begin{cases} \emptyset & \text{if } e \equiv n \\ \{\langle \text{arg}, i \rangle\} & \text{if } e \equiv \langle \text{arg}, i \rangle \\ \bigcup_{j=1}^{|e'|} \text{argvars}(e'[j], t) & \text{if } e \equiv \langle \text{ret}, i \rangle, \text{ where} \\ & t[i] \equiv \text{call*}(e') \\ \text{argvars}(e', t) & \text{if } e \equiv uop e' \\ \text{argvars}(e_1, t) \cup \text{argvars}(e_2, t) & \text{if } e \equiv e_1 \text{ bop } e_2 \end{cases}$$

On omitting commits and committing omissions:

Preventing Git metadata tampering that (re)introduces software vulnerabilities

Santiago Torres-Arias[†] Anil Kumar Ammula[‡], Reza Curtmola[‡], Justin Cappos[†]
santiago@nyu.edu aa654@njit.edu crix@njit.edu jcappos@nyu.edu

[†]*New York University, Tandon School of Engineering*

[‡]*Department of Computer Science, New Jersey Institute of Technology*

Abstract

Metadata manipulation attacks represent a new threat class directed against Version Control Systems, such as the popular Git. This type of attack provides inconsistent views of a repository state to different developers, and deceives them into performing unintended operations with often negative consequences. These include omitting security patches, merging untested code into a production branch, and even inadvertently installing software containing known vulnerabilities. To make matters worse, the attacks are subtle by nature and leave no trace after being executed.

We propose a defense scheme that mitigates these attacks by maintaining a cryptographically-signed log of relevant developer actions. By documenting the state of the repository at a particular time when an action is taken, developers are given a shared history, so irregularities are easily detected. Our prototype implementation of the scheme can be deployed immediately as it is backwards compatible and preserves current workflows and use cases for Git users. An evaluation shows that the defense adds a modest overhead while offering significantly stronger security. We performed responsible disclosure of the attacks and are working with the Git community to fix these issues in an upcoming version of Git.

1 Introduction

A Version Control System (VCS) is a crucial component of any large software development project, presenting to developers fundamental features that aid in the improvement and maintenance of a project’s codebase. These features include allowing multiple developers to collaboratively create and modify software, the ability to roll back to previous versions of the project if needed, and a documentation of all actions, thus tying changes in files to their authors. In this manner, the VCS maintains a progressive history of a project and helps ensure the integrity of the software.

Unfortunately, attackers often break into projects’ VCSs and modify the source code to compromise hosts who install this software. When this happens, an attacker can introduce vulnerable changes by adding (e.g., adding a backdoor), or removing certain elements from a project’s history (e.g., a security patch) if he or she acquires write access to the repository. By doing this, attackers are usually able to compromise a large number of hosts at once [42, 27, 13, 21, 15, 4, 45, 18, 44]. For example, the Free Software Foundation’s repository was controlled by hackers for more than two months, serving potentially backdoored versions of GNU software to millions of users [16].

The existing security measures on VCSs, such as commit signing and push certificates [19, 2], provide limited protection. While these mechanisms prevent an attacker from tampering with the contents of a file, they do not prevent an attacker from modifying the repository’s metadata. Hence, these defenses fail to protect against many impactful attacks.

In this work, we reveal several new types of attacks against Git, a popular VCS. We collectively call these attacks *metadata manipulation attacks* in which Git metadata is modified to provide inconsistent and incorrect views of the state of a repository to developers. These attacks can be thought of as *reconcilable fork attacks* because the attacker can cause a developer’s version of the repository to be inconsistent just for a finite window of time — only long enough to trick a developer into committing the wrong action — and leave no trace of the attack behind.

The impact of an attack of this nature can be substantial. By modifying the right metadata, an attacker can remove security patches, merge experimental code into a production branch, withhold changes from certain users before a release, or trick users and tools into installing a different version than the one requested to the VCS. To make matters worse, the attacker only requires a few resources to achieve his or her malicious goals.

We have submitted a vulnerability disclosure to CERT and the GitHub security team describing the following scenario: an attacker capable of performing a man-in-the-middle attack between a GitHub [3] server and a developer using pip to install Django (a popular website framework) can trick the developer into installing a vulnerable version simply by replacing one metadata file with another. Even though Git verifies that the signature in Git objects is correct, it has no mechanism to ensure it has retrieved the correct object. This type of attack enables a malicious party to strike any system that can retrieve packages from Git repositories for installation, including Node’s NPM [22], Python’s pip [11], Apache Maven [34], Rust’s cargo [35], and OCaml’s OPAM [33]. As such, it could potentially affect hundreds of thousands of client devices.

To mitigate metadata manipulation attacks, we designed and implemented a client-only, backwards-compatible solution that introduces only minimal overhead. By storing signed reference state and developer information on the server, multiple developers are able to verify and share the state of the repository at all times. When our mechanism is in place, Git metadata manipulation attacks are detected. We have presented these issues to the Git developer community and prepared patches — some of which are already integrated into Git — to fix these issues in upcoming versions of Git.

In summary, we make the following contributions:

- We identify and describe metadata manipulation attacks, a new class of attacks against Git. We show these attacks can have a significant practical impact on Git repositories.
- We design a defense scheme to combat metadata manipulation attacks by having Git developers share their perception of the repository state with their peers through a signed log that captures their history of operations.
- We implement the defense scheme and study its efficiency. An evaluation shows that it incurs a smaller storage overhead than push certificates, one of Git’s security mechanisms. If our solution is integrated in Git, the network communication and end-to-end delay overhead should be negligible. Our solution does not require server side software changes and can be used today with existing Git hosting solutions, such as GitHub, GitLab, or Bitbucket.

2 Background and related work

2.1 Overview of Git

In order to understand how Git metadata manipulation attacks take place, we must first define Git-specific terminology, as well as some usage models of the tool itself.

Git is a distributed VCS that aids in the development of software projects by giving each user a local copy of the relevant development history, and by propagating changes made by developers (or their history) between such repositories. Essential to the version history of code committed to a Git repository are `commit` objects, which contain metadata about who committed the code, when it was committed, pointers to the previous commit object, (the `parent` commit) and pointers to the objects (e.g., a file) that contain the actual committed code.

Branches serve as “pointers” to specific commit objects, and to the development history that preceded each commit. They are often used to provide conceptual separation of different histories. For example, a branch titled “`update-hash-method`” will only contain objects that modify the hash method used in a project. When a developer adds a new commit to the commit chain pointed to by a branch, the branch is moved forward.

Inside Git, branches are implemented using “reference” files, that only contain the SHA1 hash of a target commit. The same format is used for Git tags, which are meant to point to a static point in the project’s history. Both tags and branches live in the `.git/refs` folder.

Git users `commit` changes to their local repositories, and employ three main commands to propagate changes between repositories: `fetch`, to retrieve commits by other developers from a remote repository; `merge`, to merge two changesets into a single history; and `push`, to send local commits from a local repository to a remote repository. Other common commands may consist of two or more of these commands performed in conjunction (e.g., `pull` is both a `fetch` and a `merge`). Consider the following example:

Alice is working on a popular software project and is using Git to track and develop her application. Alice will probably host a “blessed” copy of her repository in one provider (e.g., GitHub or Gitlab) for everyone to clone, and from which the application will eventually be built. In her computer, she will keep a `clone` (or copy) of the remote repository to work on a new feature. To work on this feature, she will create a new `branch`, `#5-handle-unicode-filenames` that will diverge from the `master` branch from now on. As she modifies files and updates the codebase, she commits – locally – and the updates will be added to the new branch in her local clone. Once Alice is done adding the feature, she will `push` her local commits to the remote server and request a colleague to review and `merge` her changes into the `master` branch. When the changes are merged, Alice’s commits will become part of the `master` history and, on the next release cycle, they will be shipped in the new version of the software.

2.1.1 Git security features

To ensure the integrity of the repository’s history, Git incorporates several security features that provide a basic defense layer:

- Each commit object contains a cryptographic hash of its parent commit. In addition, the name of the file that contains the commit object is the cryptographic hash of the file’s contents. This creates a hash chain between commits and ensures that the history of commits cannot be altered arbitrarily without being detected.
- Users have the option to cryptographically sign a commit (a digital signature is added to the commit object) using a GPG key. This allows an auditor to unequivocally identify the user who committed code and prevents users from repudiating their commits.
- A signed certificate of the references can be pushed to a remote repository. This “push certificate” solution addresses man-in-the-middle attacks where the user and a well-behaving server can vouch for the existence of a push operation.

2.2 Related work

VCS Security. Wheeler [39] provides an overview of security issues related to software configuration management (SCM) tools. He puts forth a set of security requirements, presents several threat models (including malicious developers and compromised repositories), and enumerates solutions to address these threats. Gerwitz [17] provides a detailed description of creating and verifying Git signed commits. Signing commits allows the user to detect modifications of committed data. Git incorporates protection mechanisms, such as commit signing and commit hash chaining. Unfortunately, they do not prevent the attacks we introduce in this work.

There have been proposals to protect sensitive data from hostile servers by incorporating secrecy into both centralized and distributed version control systems [1, 29]. Shirey et al. [32] analyzes the performance trade-offs of two open source Git encryption implementations. Secrecy from the server might be desirable in certain scenarios, but it is orthogonal to our goals in this work.

The “push certificate” mechanism, introduced in version 2.2.0 of Git, allows a user to digitally sign the reference that points to a pushed object. However, push certificates do not protect against most of the attacks we describe in this work. Furthermore, push certificates were designed for out-of-band auditing (*i.e.* they are not integrated into the usual workflow of Git and need to be fetched and verified by a trusted third party using out-of-band mechanisms). As a result, push certificates are rarely used in practice.

Fork Consistency. A problem that could arise in remote storage used for collaborative purposes is when the un-

trusted storage server hides updates performed by one group of users from another. In other words, the server equivocates and presents different views of the history of operations to different groups of users. The *fork consistency* property seeks to address this attack by forcing a server that has forked two groups in this way to continue this deception. Otherwise, the attack will be detected as soon as one group sees an operation performed by the other group after the moment the fork occurred.

SUNDR [26] provides fork consistency for a network file system that stores data on untrusted servers. In SUNDR, users sign statements about the complete state of all the files and exchange these statements through the untrusted server. SPORC [14] is a framework for building collaborative applications with untrusted servers that achieves fork* consistency (*i.e.*, a weaker variant of fork consistency). Our solution seeks to achieve a similar property and shares similarities with SUNDR in that Git users leverage the actual Git repository to create and share signed statements about the state of the repository. However, the intricacies and usage model of a VCS system like Git impose a different set of constraints.

Other work, such as Depot [28], focuses on recovering from forks in an automatic fashion (*i.e.*, not only detecting forks, but also repairing after they are detected). Our focus is on detecting the metadata manipulation attacks, after which the affected users can perform a manual rollback procedure to a safe point.

Caelus [25] seeks to provide the same declared history of operations to all clients of a distributed key-value cloud store. Caelus assumes that no external communication channel exists between clients, and requires them to periodically attest to the order and timing of operations by writing a signed statement to the cloud every few seconds. The attestation schedule must be pre-defined and must be known to all clients. Our setting is different, since Git developers usually communicate through multiple channels; moreover, a typical team of Git developers cannot be expected to conform to such an attestation policy in practice.

3 Threat model and security guarantees

We make the following assumptions about the threat model our scheme is designed to protect against:

- Developers use the existing Git signing mechanisms whenever performing an operation in Git to stop an attacker from tampering with files.
- An attacker cannot compromise a developer’s key or get other developers to accept that a key controlled by an attacker belongs to a legitimate developer. Alternatively, should an attacker control such a key (*e.g.*, an insider attack), he or she may not

want to have an attack attributed to him- or herself and would thus be unwilling to sign data they have tampered with using their key.

- The attacker can read and modify any files on the repository, either directly (*i.e.* a compromised repository or a malicious developer) or indirectly (*i.e.*, through MITM attacks and using Git’s interface to trick honest users into doing it).
- The attacker does not want to alert developers that an attack has occurred. This may lead to out-of-band mechanisms to validate the attacked repository [30].

This threat model covers a few common attack scenarios. First of all, an attacker could have compromised a software repository, an unfortunately common occurrence [42, 27, 13, 21, 15, 4, 45, 18, 44, 16]. Even if the repository is not compromised, an attacker could act as a man-in-the-middle by intercepting traffic destined for the repository (e.g., by forging SSL certificates [23, 31, 8, 37, 43, 7, 41, 6, 38]). However, an attacker is not limited to these strategies. As we will show later, a malicious developer can perform many of the same attacks *without using their signing key*. This means that it is feasible for a developer inside an organization to launch these attacks and not be detected.

Note that in all cases, the developers have known signing keys to commit, push, and verify information.

3.1 Security guarantees

Answering to this threat model, the goal of a successful defensive system should be to enforce the following:

- **Prevent modification of committed data:** If a file is committed, an attacker should not be able to modify the file’s contents without being detected.
- **Ensure consistent repository state:** All developers using a repository should see the same state. The repository should not be able to equivocate and provide different commits to different developers.
- **Ensure repository state freshness:** The repository should provide the latest commits to each developer.

As we will show later, Git’s existing security mechanisms fail to handle the last two properties. The existing signing mechanism for Git does enable developers to detect modification of committed data, because the changed data will not be correctly signed. However, due to weaknesses in handling the other properties, an attacker can omit security patches, merge experimental features into production, or serve versions of software with known vulnerabilities.

An attacker is successful if he or she is able to break any of these properties without being detected by the developers. So, an attacker who controls the repository could block a developer from pushing an update by pretending the repository is offline. However, since the developer receives an error, it is obvious that an attack is occurring and therefore is easy to detect. Similarly, this also precludes *irreconcilable fork attacks* where two sets of developers must be permanently segregated from that point forward. Since developers typically communicate through multiple channels, such as issue trackers, email, and task management software, it will quickly become apparent that fixes are not being merged into the master branch. (Most projects have a tightly integrated team, usually a single person, who integrates changes into the master branch, which further ensures this attack will be caught.) For these reasons, we do not focus on attacks that involve a trivial denial of service or an irreconcilable fork because they are easy to detect in practice.

4 Metadata manipulation attacks

Even when developers use Git commit signing, there is still a substantial attack surface. We have identified a new class of attacks that involve manipulation of Git metadata stored in the `.git/refs` directory of each repository. We emphasize that, unlike Git commits that can be cryptographically signed, there are no mechanisms in Git to protect this metadata. As such, the metadata can be tampered with to cause developers to perceive different states of the repository, which can coerce or trick them into performing unintended operations in the repository. We also note that a solution that simply requires users to sign Git metadata has serious limitations (as described in Sec. 5.2).

Unlike many systems where equivocation is likely to be noticed immediately by participants, Git’s use of branches hides different views of the repository from developers. In many development environments, developers only have copies of branches that they are working on stored locally on their system, which makes it easy for a malicious repository to equivocate and show different views to different developers.

In Git, a branch is represented by a file that contains the SHA1 checksum of a commit object (under benign circumstances, this object is the latest commit on that branch). We will refer to such files as *branch references*. All the branch references are stored in the directory `.git/refs/heads/`, with the name of the branch as the filename. For example, a branch “hotfix” is represented by the file `.git/refs/heads/hotfix`.

We discovered that it is straightforward for an attacker to manipulate information about branches by simply changing contents in a reference file to point to any

other commit object. *Modifying the branch reference can be easily performed with a text editor and requires no sophistication.* Specifically, we show three approaches to achieve this, all of them being captured by our adversarial model. *First*, an attacker who has compromised a Git repository and has write access to it, can directly modify the metadata files. *Second*, an attacker can perform an MITM attack by temporarily redirecting a victim’s traffic to a fake repository serving tampered metadata, and then reestablishing traffic so the victim propagates the vulnerable changes to the genuine repository (in Appendix A, we describe a proof-of-concept attack against GitHub based on this approach). *Third*, a malicious developer can take advantage of the fact that Git metadata is synchronized between local and remote repositories. The developer manipulates the Git metadata in her local repository, which is then propagated to the (main) remote repository.

It is also possible to extend these attacks for Git tags. Although a Git tag is technically a Git tag object that can be signed the same way as a commit object, an attacker can target the *reference* pointing to a tag. Tag references are stored in the directory `.git/refs/tags/` and work similarly to branch references, in that they are primarily a file containing the SHA1 of a Git tag object that points to a Git commit object. Although Git tags are conceptually different — they only represent a fixed point (e.g., a major release version) in the projects history — they can be exploited in the same way, because Git has no mechanism to protect either branch or tag references.

We have validated the attacks against a standard Git server and also the GitHub, GitLab and other popular Git hosting services.

Based on their effect on the state of the repository, we identify three types of metadata manipulation attacks:

- **Teleport Attacks:** These attacks modify a Git reference so that it points to an arbitrary object, different from the one originally intended. The reference can be a branch reference or a tag reference.
- **Rollback Attacks:** These attacks modify a Git branch reference so that it points to an older commit object from the same branch, thus providing clients with a view in which one or more of the latest branch commits are missing.
- **Deletion Attacks:** These attacks remove branch or tag references, which in turns leads to the complete removal of an entire branch, or removal of an entire release referred to by a tag.

We use the following setup to present the details of these attacks. A Git server is hosting the main repository

and several developers who have their own local repositories have permission to fetch/push from/to any branch of the main repository, including the master branch. For commit objects, we use a naming convention that captures the temporal ordering of the commits. For example, if a repository has commits C0, C1, C2, this means that they were committed in the order C0, C1, C2.

4.1 Teleport attacks

We identified two teleport attacks: *branch teleport* and *tag teleport* attacks.

Branch Teleport Attacks. These attacks modify the branch reference so that it points to an arbitrary commit object on a different branch. Although we illustrate the attacks for the master branch, they are applicable to any branch, since none of the branch reference metadata is protected.

Fig. 1(a) shows the initial state of the main Git repository, which contains two branches, “master” and “feature.” The local repository of developer 1 is in the same state as shown in Fig. 1(a). The “feature” branch implements a new feature and contains one commit, C2. The code in C2 corresponds to an unstable, potentially-vulnerable version that needs to be tested more thoroughly before being integrated into the master branch. Commit C1 is the head of the master branch. This means that the file `.git/refs/heads/master` contains the SHA1 hash of the C1 commit object.

After developer 2 pulls from the master branch of the main repository (Fig. 1(b)), the attacker changes the master branch to point to commit C2 (Fig. 1(c)). The attacker does this by simply changing the contents of the file `.git/refs/heads/master` to the SHA1 hash of the C2 commit. Any developer who clones the repository or fetches from the master branch at this point in time will be provided with the incorrect repository state, as shown in Fig. 1(c). For example, developer 2, who committed C3 into his local repository (Fig. 1(d)), now wants to push this change to the main repository. Developer 2 is notified that there were changes on the master branch since his last fetch, and needs to pull these changes. As a result, a merge commit C4 occurs between C3 and C2 in the local repository of developer 2, as shown in Fig. 1(e). The main repository looks like Fig 1(e) after developer 2 pushes his changes. If developer 1 then pulls changes from the main repository, all three repositories will appear like Fig 1(e).

Normally, the master branch should contain software that was thoroughly tested and properly audited. However, in this incorrect history, the master branch incorporates commit C2, which was in a experimental feature branch and may contain bugs. The attacker tricked a developer into performing an action that was never in-

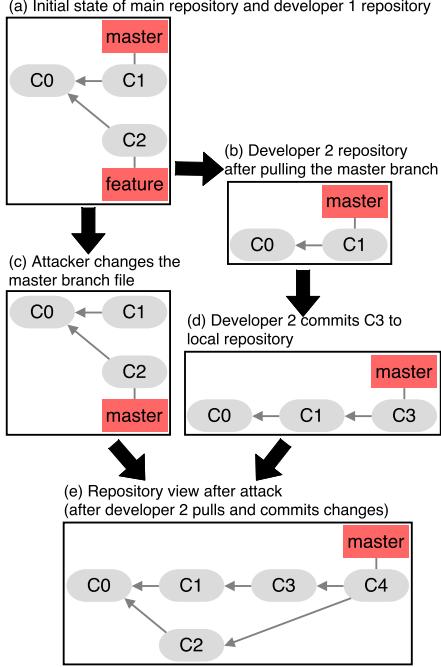


Figure 1: The Branch Teleport attack

tended, and none of the two developers are aware that the attack took place.

Tag Teleport Attacks. These attacks modify a tag reference so that it points to an arbitrary tag object. Surprisingly, a tag reference can also be made to point to a commit object, and Git commands will still work.

One can verify whether a tag is both signed and a valid tag object by using the `git tag --verify` command. However, if an attacker were to modify a tag reference to point to an older tag (e.g., if the tag for release 1.1 is replaced by the tag for the vulnerable release 1.0), the verification command is successful.

Modifying tag metadata could be especially impactful for automated systems that rely on tags to build/test and release versions of software [36, 20, 10, 12]. Furthermore, package managers such as Python’s pip, Ruby’s RubyGEMS, and Node’s NPM, among many others support the installation of software from public Git repositories and tags. Finally, Git submodules are also vulnerable, as they automatically track a tag (or branch). If a build dependency is included in a project as a part of the submodule, a package might be vulnerable via an underlying library.

4.2 Rollback attacks

These attacks modify a Git branch reference so that it points to an older commit object from the same branch. This gives clients a view in which one or more of the latest branch commits are missing. The attacker can cause commits to be missing on a permanent or on a temporary basis.

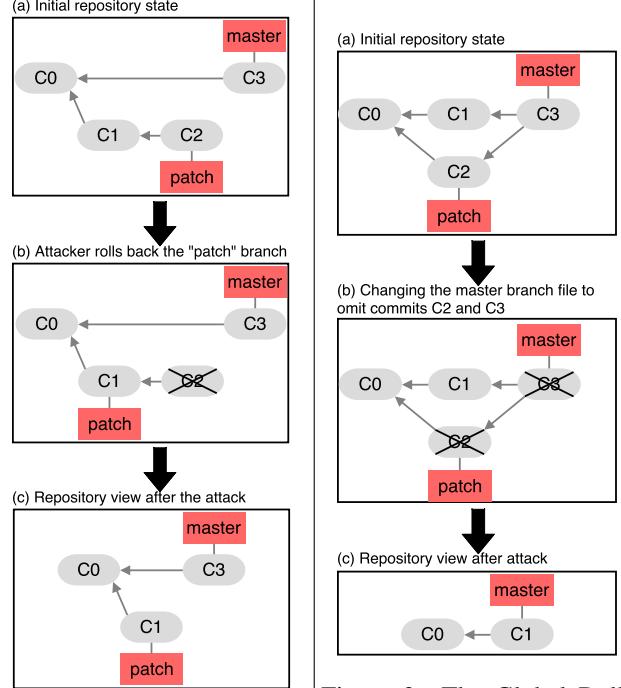


Figure 2: The Branch Rollback attack

4.2.1 Permanent rollback attacks

Based on the nature of the commits removed, we separate permanent rollback attacks in two groups: Branch Rollback attacks and Global Rollback attacks.

Branch Rollback Attacks. Consider the repository shown in Fig. 2(a), in which the order of the commits is C0, C1, C2, C3. Commits C0 and C3 are in the master branch, and commits C1 and C2 are security patches in a “patch” branch. The attacker rolls back the patch branch by making the head of such branch point to commit C1, as shown in Fig. 2(b). This can be done by simply replacing the contents of the file `.git/refs/heads/patch` with the SHA1 hash of the C1 commit. As a result, all developers that pull from the main repository after this attack will see the state shown in Fig. 2(c), in which commit C2 (that contains a security patch) has been omitted.

Note that the attack can also be used to omit commits on any branch, including commits in the master branch.

Global Rollback Attacks. As opposed to a Branch Rollback attack, which removes commits that happened prior to one that remains visible, in a Global Rollback attack, no commits remain visible after the commits that are removed. In other words, the attacker removes one or more commits that were added last to the repository.

Consider the initial state of a Git repository as illustrated in Fig. 3(a), in which C2 is a commit that fixes a security bug and has been merged into the master

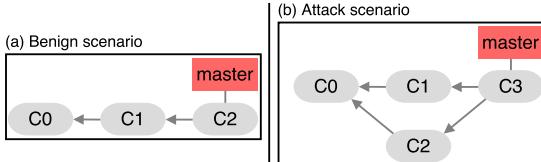


Figure 4: The Effort Duplication attack

branch. The file `.git/refs/heads/master` contains the SHA1 hash of the C3 commit object.

By simply changing the contents of the file `.git/refs/heads/master` to the SHA1 hash of the C1 commit, the attacker forges a state in which the repository contains the history of commits depicted in Fig. 3(b). This effectively removes commits C2 and C3 from the project’s history, and a developer who now clones the project will get a history of commits as shown in Fig. 3(c). This incorrect history does not contain the commit C2 that fixed the security bug.

Note that the Global Rollback attack *removed the latest two commits from the repository*. This is different than the effect of a Branch Rollback attack which *removes one or more commits that happened before a commit that remains visible*.

4.2.2 Temporary rollback attacks

Effort Duplication Attacks. The Effort Duplication attack is a variation of the Global Rollback attack, in which the attacker temporarily removes commits from the repository. This might cause developers to unknowingly duplicate coding efforts that exist in the removed commits.

Consider a main Git repository with just a master branch which contains only one commit C0. Two developers D1 and D2 have pulled from the main repository, so their local repositories also contain C0. After the following sequence of actions by D1 and D2, the repository should look as shown in Fig. 4(a):

1. D1 commits C1 to her local repository & pushes to the main repository.
2. D2 pulls from the main repository.
3. D2 commits C2 to her local repository & pushes to the main repository.

However, when D2 pulls in step 2, the attacker can temporarily withhold commit C1, keeping D2 unaware of the changes in C1. As a result, D2 works on changes that already exist in C1. The following attack scenario results in a repository shown in Fig. 4(b):

1. D1 commits C1 to her local repository & pushes to the main repository.
2. D2 pulls from the main repository, but the attacker withholds C1. Thus, D2 thinks there are no changes.
3. D2 makes changes on top of C0 and commits these changes in her local repository as commit C2. C2 duplicates (some or all of) D1’s coding effort in C1.

4. D2 tries to push changes to the main repository. This time, the attacker presents C1 to D2 (these are the changes that were withheld in step 2). Thus, D2 has to first pull changes before pushing.
5. D2 pulls changes from the main repository, and this results in a merge commit C3 between C1 and C2. As part of the merge, the developer has to solve any merge conflicts that appear from the code duplication between C1 and C2.

In this case, D2 re-did a lot of D1’s work because D1’s commit C1 was withheld by the attacker. Note that unlike a Global Rollback attack, in which commits are removed permanently from the repository, in the Effort Duplication attack commits are just removed temporarily. This is a more subtle attack, since the final state of the repository is the same for both the benign and attack cases. The effect of applying commits C1 and C2 in Fig. 4(a) on the files in the repository is the same as applying commits C1, C2, C3 in Fig. 4(b). However, D2 unknowingly (and unnecessarily) duplicated D1’s coding effort, which may have negative economic consequences. Adding to this, an attacker can slow down developers of a specific project (e.g., a competitor’s project) by delivering previously-withheld changes to them when they will cause merge conflicts and hamper their development progress.

4.3 Reference deletion attacks

Since the branch metadata is not protected, the attacker can hide an entire branch from the repository by removing a branch reference. Similarly, since the tag metadata is not protected, the attacker can remove a tag reference in order to hide a release from the repository history.

When an attacker performs a reference deletion attack, only the users who previously held a copy of the reference will be able to know of its existence. If this is not the case, a developer would be oblivious of the fact that other developers have worked on the deleted branch (similar to a fork attack), or be tricked into retrieving another version if the target tag is not available. Furthermore, some projects track work in progress by tying branch names to numbers in their issue tracker [9], so two developers could be tricked into working on the same issue by hiding a branch (similar to an effort duplication attack).

4.4 Summary of attacks

Metadata manipulation attacks may lead to inconsistent and incorrect views of the repository and also to corruption and loss of data. Ultimately, this will lead to merge conflicts, omission of bug fixes, merging experimental code into a production branch, or withholding changes from certain users before a release. All of these are problems that can impact the security and stability of

the system as a whole. Table 1 summarizes the attacks impact.

Attack	Impact
Branch Teleport	Buggy code inclusion
Branch Rollback	Critical code omission
Global Rollback	Critical code omission
Effort Duplication	Coding effort duplicated
Tag Rollback	Older version retrieved

Table 1: Impact of metadata manipulation attacks.

5 Defense framework

5.1 Design goals for a defense scheme

We designed our defense scheme against metadata manipulation attacks with the following goals in mind:

Design Goal 1 (DG1): Achieve the security goals stated in Sec. 3.1. That is, prevent modification of committed data, ensure a consistent repository state, and ensure repository state freshness.

Design Goal 2 (DG2): Preserve (as much as possible) current workflows and actions that are commonly used by developers, in order to facilitate a seamless adoption.

Design Goal 3 (DG3): Maintain compatibility with existing Git implementations. For example, Git has limited functionality when dealing with concurrency issues in a multi-user setting: it only allows atomic push of multiple branches and tags after version 2.4. Following Git’s design philosophy, backwards compatibility is paramount; a server running the latest Git version (*i.e.*, 2.9.0) can be cloned by a client with version 1.7.

5.2 Why binding references with Git objects is not enough

Adding reference information (*i.e.*, branch and tag names) inside the commit object might seem like a sufficient defense against metadata manipulation attacks. This would bind a commit to a reference and prevent an attacker from claiming that a commit object referred to in a reference belongs somewhere else.

Unfortunately, this simple approach has important drawbacks. It does not meet our *DG1* because it does not defend against rollback and effort duplication attacks. Furthermore, adding new reference information in a commit object requires updating an existing commit object. When this happens, the SHA-1 hash of the commit object will change, and the change will propagate to all new objects in the history. In other words, when a new branch is created and bound to a commit far earlier in the history, all commit objects need to be rewritten and, thus, sent back to the remote repository, which could add substantial computational and network overhead.

5.3 Our defense scheme

The fundamental cause of metadata manipulation attacks is that the server can respond to users’ fetches with an incorrect state and history of the main repository that they cannot verify. For example, the server can falsely claim that a branch points to a commit that was never on that branch or to a commit that was the location of that branch in an earlier version. Or, the server can falsely claim that the reference of a tag object points to an older tag.

In order to stop the server from falsely claiming an incorrect state of the repository, we propose that every Git user must include additional information vouching for their perceived repository state during a push or a fetch operation. To achieve this, we include two pieces of additional information on the repository:

- First, upon every push, users must append a *push entry* to a *Reference State Log (RSL)* (Sec. 5.3.1). By validating new entries in this log with each push and fetch operation, we can prevent teleport, permanent rollback, and deletion attacks.
- Second, when a Git user performs a fetch operation and receives a new version of files from the repository, the user places a random value into a *fetch nonce bag* (Sec. 5.3.2). If the Git user does not receive file updates when fetching, the user replaces her value in the bag with a new one. The bag serves to protect against temporary rollback attacks.

During our descriptions, we assume that a trusted key-chain is distributed among all developers along with the RSL. There are tools available to automate this process [24, 5], and the RSL itself can also be used to distribute trust (we elaborate more on this in Sec 6.1).

5.3.1 The Reference State Log (RSL)

For a developer to prevent the server from equivocating on the location of the references, the developer will sign a *push entry*, vouching for the location of the references at the time of a push. To do this, she must execute the `Secure_push` procedure, which has the following steps:

First, the remote RSL is retrieved, validated, and checked for the presence of new push entries (lines 3–11). If the RSL is valid and no push entries were added, a new RSL push entry is created (lines 13–14). The newly created entry will contain: (1) the new location of the reference being pushed; (2) the nonces from the fetch nonce bag; (3) a hash of the previous push entry; and (4) the developer’s signature over the newly created push entry. The newly created entry is then appended to the RSL (line 16), and the `nonce_bag` is cleared (line 15).

Once this is done, the remote RSL must be updated and local changes must be pushed to the remote reposi-

PROCEDURE: Secure_push

Input: LocalRSL; related_commits; pushed reference X
Output: result: (success/fail/invalid)

```
1: repeat
2:   result ← fail
3:   (RemoteRSL, nonce_bag) =
   Retrieve_RSL_and_nonce_bag_from_remote_repo
4:   if (RSL_Validate(RemoteRSL, nonce_bag) == false)
   then
5:     // Retrieved RemoteRSL is invalid
6:     // Must take necessary actions!
7:     return invalid
8:   if (new push entries for reference X in RemoteRSL) then
9:     // Remote repository contains changes
10:    // User must fetch changes and then retry
11:    return fail
12:   else
13:     prev_hash = hash.last_push_entry(RemoteRSL)
14:     new_RSL_Entry = create_push_Entry(prev_hash,
   nonce_bag, X)
15:     nonce_bag.clear()
16:     RemoteRSL.addEntry(new_RSL_Entry)
17:     result = Store_in_remote_repo(RemoteRSL,
   nonce_bag)
18:   if (result == success) then
19:     // The remote RSL has no new entries
20:     push related_commits
21:     LocalRSL = RemoteRSL
22:     return success
23: until (result == success)
```

tory (lines 17-20). Notice that these steps are performed under a loop, because other developers might be pushing, which is not an atomic operation in older versions of Git (this is required to meet *DG3*).

Depending on the result of the Secure_push procedure, a developer's actions correspond to the following:

- **success:** the push is successful. No further actions are required from the user (line 22).
- **fail:** the push fails because there are changes in the remote repository that must first be fetched and merged locally before the user's changes can be pushed (line 11).
- **invalid:** the RSL validation has failed. The algorithm detects a potential attack and notifies the user, who must then take appropriate measures (line 7).

Note that these actions mirror a user's actions in the case of a regular Git push operation, as suggested by *DG2*. By doing this, we effectively follow the existing Git workflows while providing better security guarantees at the same time.

5.3.2 TheNonce Bag

When retrieving the changes from a remote repository, a developer must also record her perceived state of the repository. Our scheme requires that all the user fetches be recorded in the form of a fetch *nonce bag*, *i.e.*, an unordered list of nonces. Each nonce is a random number that corresponds to a fetch from the main repository. Every time a user fetches from the main repository, she updates the nonce bag. If the user has not fetched since the last push, then she generates a new nonce and adds it to the nonce bag; otherwise, the user replaces her nonce in the nonce bag with a new nonce.

Each nonce in the nonce bag serves as a proof that a user was presented a certain RSL, preventing the server from executing an Effort Duplication attack and providing repository freshness as per *DG1*. To fetch the changes from the remote repository, a developer must execute the Secure_fetch procedure.

The first steps of the Secure_fetch procedure consist of retrieving the remote RSL, performing a regular git fetch, and ensuring that the latest push entry in the RSL points to a valid object in the newly-fetched reference (lines 4-11). Note that this check is performed inside a loop because push operations are not atomic in older versions of Git (lines 2-12). A user only needs to retrieve the entries which are new in the remote RSL and are not present in the local version of the RSL.

If this check is successful, the nonce bag must be updated and stored at the remote repository (lines 14-20). Note that all these steps are also in a loop because other developers might update the RSL or the nonce bag since it was last retrieved (lines 1-21).

Finally, the RSL is further validated for consistency (line 22), and the local RSL is updated. We chose to validate the RSL at the end of Secure_fetch and outside of the loop in order to optimize for the most common case. Once Secure_fetch is successfully executed, a developer can be confident that the state of the repository she fetched is consistent with her peers. Otherwise, the user could be the victim of one of the attacks in Sec. 4.

5.3.3 RSL validation

The RSL_Validate routine is used in Secure_push and Secure_fetch to ensure the presented RSL is valid. The aim of this routine is to check that push entries in a given RSL are correctly linked to each other, that they are signed by trusted developers, and that nonces corresponding to a user's fetches are correctly incorporated into the RSL.

First, the procedure checks that the nonce corresponding to the user's last fetch appears either in the nonce bag or was incorporated into the right push entry (*i.e.*, the first new push entry of the remote RSL) (lines 1-2). The algorithm then checks if the new push RSL entries from the

PROCEDURE: Secure_fetch**Input:** reference X to be fetched**Output:** result: (success/invalid)

```
1: repeat
2:   store_success ← false
3:   repeat
4:     (RemoteRSL, nonce_bag) =
      Retrieve_RSL_and_nonce_bag_from_remote_repo()
5:     fetch_success ← false
6:     // This is a regular “Git fetch” command.
7:     // Branch X’s reference is copied to FETCH_HEAD
8:     fetch reference X
9:     C ← RemoteRSL.latestPush(X).refPointer
10:    if (C == FETCH_HEAD) then
11:      fetch_success ← true
12:    until (fetch_success == true)
13:    // Update the nonce bag
14:    if NONCE in nonce_bag then
15:      nonce_bag.remove(NONCE)
16:    save_random_nonce_locally(NONCE)
17:    nonce_bag.add(NONCE)
18:    // Storing the nonce bag at the remote repository
19:    // might fail due to concurrency issues
20:    store_success = Store_in_remote_repo(nonce_bag)
21:    until (store_success == true)
22:    if (RSL_Validate(RemoteRSL, nonce_bag) == false)
        then
23:      // Retrieved RemoteRSL is invalid
24:      // Must take necessary actions!
25:      return invalid
26:    else
27:      LocalRSL = RemoteRSL
28:      return success
```

remote RSL are correctly linked to each other and that the first new remote push entry is correctly linked to the last push entry of the local RSL (the check is based on the prev_hash field) (lines 5-9). Finally, the signature on the last RSL push entry is verified to ensure it was signed by a trusted developer; since all RSL entries are correctly linked, only the last entry signature needs to be verified.

How to handle misbehavior? If the RSL validation fails due to a misbehaving server, the user should compare the local RSL with the remote RSL retrieved from the remote repository and determine a safe point up to which the two are consistent. The users will then manually roll back the local and remote repositories to that safe point, and decide whether or not to continue trusting the remote repository.

6 Discussion

6.1 Trust and revoke entries

Developers’ keys may be distributed using *trust/revoke RSL entries*. To use these entries, the

PROCEDURE: RSL_Validate**Input:** LocalRSL (RSL in the local repository); RemoteRSL; nonce_bag**Output:** true or false

```
1: if (NONCE not in nonce_bag) and (NONCE not in RemoteRSL.push_after(LocalRSL)) then
2:   return false
3:   // Verify that the ensuing entries are valid
4:   prev_hash = hash_last_push_entry(LocalRSL)
5:   for new_push_entry in RemoteRSL do
6:     if new_push_entry.prev_hash != prev_hash then
7:       // The RSL entries are not linked correctly
8:       return false
9:     prev_hash = hash(new_push_entry)
10:    if verify_signature(RemoteRSL.latest_push) == false then
11:      // this RSL is not signed by a trusted developer
12:      return false
13:    return true
```

repository is initialized with an authoritative root of trust (usually a core developer) who will add further entries of new developers in the group. Once developers’ public keys are added to the RSL, they are allowed to add other trust entries.

A trust entry contains information about the new developer (i.e., username and email), her public key, a hash of the previous push entry and a signature of the entry by a trusted developer. Revocation entries are similar in that they contain the key-id of the untrusted developer, the hash of the push entry, and the signature of the developer revoking trust.

6.2 Security analysis

Our defense scheme fulfills the properties described in Sec. 3.1 as follows:

- Prevent modification of committed data: The existing signing mechanism for Git handles this well. Also, RSL entries are digitally signed and chained with each other, so unauthorized modifications will be detected.
- Ensure consistent repository state: The RSL provides a consistent view of the repository that is shared by all developers.
- Ensure repository state freshness: The Nonce Bag provides repository state freshness because an attacker cannot replay nonces in the Nonce Bag. Also, if no newer push entries are provided by the repository, then the attack becomes a fork attack.

The attacks described in Section 4 are prevented because, after performing the attack, the server cannot provide a valid RSL that matches the current repository

	Possible attacks	Time window of attack	Vulnerable commit objects
Commit signing	all attacks	Anytime	Any object
RSL (full adoption)	no attacks	None	No object
RSL (partial adoption)	all attacks	After the latest RSL entry and before the next RSL entry	Objects added after the latest RSL entry

Table 2: Security guarantees offered by different adoption levels of the defense scheme

state. Since she does not control any of the developers’ keys, she can not forge a signature for a spurious RSL entry. As a result, a user who fetches from the main repository after the attack will notice the discrepancy between the RSL and the repository state that was presented to her. Each metadata manipulation attack would be detected as follows:

- *Branch Teleport and Deletion Attacks:* When this attack is performed, there is no mention of this branch pointing to such a commit, and the RSL validation procedure will fail.
- *Branch/Global Rollback and Tag Teleport Attacks:* These attacks can be detected because the latest entry in the RSL that corresponds to that branch points to the commit removed and the RSL validation procedure will fail.
- An attacker can attempt to remove the latest entries on the RSL so that the attacks remain undetected. However, after this moment, the server would need to consistently provide an incorrect view of the RSL to the target user, which would result in a fork attack. Finally, the attacker cannot remove RSL entries in between because these entries are chained using the previous hash field. Thus, the signature verification would fail if this field is modified.
- *Effort Duplication Attack:* This attack will result in a fork attack because the RSL created by the user requesting the commit will contain a proof about this request in the form of a nonce that has been incorporated into an RSL push entry or is still in the Nonce Bag. Any ensuing RSL push entry that was withheld from the user will not contain the user’s nonce.

6.3 Partial adoption of defense scheme

It is possible that not all developers in a Git repository use our solution. This can happen when, for example, a user has not configured the Git client to sign and update the RSL. When this is the case, the security properties of the RSL change.

To study the properties of using the RSL when not everyone is using the defense, we will define a commit object as a “*secure commit*” or an “*insecure commit*. The former will be commits made by users who employ our defense, while the latter are made by users who do not

use our defense (*i.e.*, they only use the Git commit signing mechanism). Consider that supporting partial adoption requires changing the validation during fetches to consider commits that are descendants of the latest secure commit, for users might push to branches without using the defense. For simplicity, we do not allow users to reset branches if they are not using the defense.

Compared to commit signing only, when our scheme is adopted by only some users, a user who writes an RSL entry might unwittingly attest to the insecure commits made by other users after the latest secure commit. However, this situation still provides a valuable advantage because the attacker’s window to execute Metadata Manipulation attacks is limited in time. That is, when our defense is not used at all, an attacker can execute Metadata Manipulation attacks on any commits in the repository, (e.g., the attacker can target a forgotten branch located early in the history). This is not possible with our scheme, where an attacker can only attack the commit objects added after the latest RSL entry for that branch. The differences between the three alternatives are summarized in Table 2.

6.4 Comparison with other defenses

In Table 3, we examine the protections offered by other defense schemes against metadata manipulation attacks. Specifically, we studied how Git commit signing, Git’s push certificate solution, and our solution (listed as RSL) fare against the attacks presented in this paper, as well as other usability aspects that may impact adoption.

Feature	Commit signing	Push certificate	RSL
Commit Tampering	✓	✓	✓
Branch Teleport	X	✓	✓
Branch Rollback	X	X	✓
Global Rollback	X	X	✓
Effort Duplication	X	X	✓
Tag Rollback	X	✓	✓
Minimum Git Version	1.7.9	2.2.0	1.7.9
Distribution Mechanism	in-band	(no default) or Additional server	in-band

Table 3: Comparison of defense schemes against Git metadata manipulation attacks. A ✓ indicates the attack is prevented.

As we can see, Git commit signing does not protect against the vast majority of attacks presented in

this paper. Also, Git’s push certificate solution provides a greater degree of protection, but still fails to protect against all rollback and effort duplication attacks. This is primarily because (1) a server could misbehave and not provide the certificates (there is no default distribution mechanism), and (2) a server can replay old push certificates along with an old history. Basically, this solution assumes a well-behaving server hosting push certificates.

In contrast, our solution protects against all attacks presented in Table 3. In addition to this, our solution presents an in-band distribution mechanism that does not rely on a trusted server in the same way that commit signing does. Lastly, we can see that our solution can be used today, because it does not require newer versions of Git on the client and requires no changes on the server, which allows for deployment in popular Git hosting platforms such as GitHub and Gitlab.

7 Implementation and evaluation

We have implemented a prototype for our defense scheme. This section provides implementation details and presents our experimental performance results.

7.1 Implementation

To implement our defense scheme, we leveraged *Git custom commands* to replace the push and fetch commands, and implemented the RSL as a separate branch inside the repository itself. To start using the defense, a user is only required to install two additional bash scripts and use them in lieu of the regular fetch and push commands. Our client scripts consist of less than a hundred (86) lines of code, and there is no need to install anything on the server.

RSL and Nonce Bag. We implemented the RSL in a detached branch of the repository, named “RSL.” Each RSL entry is stored as a Git commit object, with the entry’s information encoded in the commit message. We store each entry in a separate commit object to leverage Git’s pack protocol, which only sends objects if they are missing in the local client. Encoding the Git commit objects is also convenient because computing the previous hash field is done automatically.

We also represent the Nonce Bag as a Git commit object at the head of the RSL branch. When a nonce is added or updated, a new commit object with the nonces replaces the previous nonce bag, and its parent is set to the latest RSL entry. When a new RSL entry is added, the commit containing the nonce bag is garbage collected by Git because the RSL branch cannot reach it anymore.

When `securepush` is executed, the script first fetches and verifies the remote RSL branch. If verification is successful, it then creates an RSL entry by issuing a new commit object with a NULL tree (i.e., no local

Field	Description
Branch	Target branch name
HEAD	Branch location (target commit)
PREV_HASH	Hash over the previous RSL entry
Signature	Digital signature over RSL entry

Table 4: RSL push entry fields.

files), and a message consisting of the fields described in Table 4. After the new commit object with the RSL push entry is created, the RSL branch is pushed to the remote repository along with the target branch.

A `securefetch` invocation will fetch the RSL branch to update or add the random nonce in the Nonce Bag. If a nonce was already added to the commit object (with a NULL tree also), it will be amended with the replaced nonce. In order to keep track of the nonce and the commit object to which it belongs, two files are stored locally: `NONCE_HEAD`, which contains the reference of the Nonce Bag in the RSL branch, and `NONCE`, which contains the value of the nonce stored in it.

Atomicity of Git operations. The `securepush` and `securefetch` operations require fetching and/or pushing of the RSL branch in addition to the pushing/fetching to/from the target branch. Git does not support atomic fetch of multiple branches, and only supports atomic push of multiple branches after version 2.4.0¹.

In order to ensure backwards compatibility, we designed our solution without considering the existence of atomic operations. Unfortunately, the lack of atomic push opens the possibility of a DoS attack that exploits the ‘repeat’ loop in `Secure_fetch` (lines 3-12), that makes the algorithm loop endlessly. This could happen if a user executes `Secure_push` and is interrupted after pushing a new RSL entry, but before pushing the target branch (e.g., caused by a network failure). Also, a malicious user may provide an updated RSL, but an outdated history for that branch. However, this issue can be easily solved if the loop is set to be repeated only a finite number of times before notifying the user of a potential DoS attempt.

If atomic push for multiple branches is available, the `Secure_push` procedure can be simplified by replacing lines 17-22 with a single push. Availability of atomic push also eliminates the possibility of the endless loop mentioned above.

7.2 Experimental evaluation

Experimental Setup. We conducted experiments using a local Git client and the GitHub server that hosted the main repository. The client was running on an Intel Core i7 system with two CPUs and 8 GB RAM. The client software consisted of OS X 10.11.2, with Git 2.6.2 and

¹Note that **both Git client and server** must be at least version 2.4.0 in order to support atomic push.

the GnuPG 2.1.10 library for 1024-bit DSA signatures.

Our goal was to evaluate the overhead introduced by our defense scheme. Specifically, we want to determine the additional storage induced by the RSL, and the additional end-to-end delay induced by our `securefetch` and `secure push` operations. For this, we used the five most popular GitHub repositories²: `bootstrap`, `angular.js`, `d3`, `jQuery`, `oh-my-zsh`. We will refer to these as R1, R2, R3, R4, and R5, respectively. In the experiments, we only considered the commits in the master branch of the these repositories. Table 5 provides details about these repositories.

Repo.	Number of commits	Number of pushes	Repo. size	Repo. size with signed commits
R1	11,666	1,345	73.04	78.85
R2	7,521	26	66.96	69.79
R3	3,510	255	32.91	34.65
R4	6,031	194	15.79	19.98
R5	3,841	1,170	3.52	4.01

Table 5: The repositories used for evaluation (sizes are in MBs).

We used the repositories with signed commits as the baseline for the evaluation. We evaluated three defense schemes:

- Our defense: This is our proposed defense scheme.
- Our defense (light): A light version of our defense scheme, which does not use the nonce bag to keep track of user fetches. This scheme sacrifices protection against Effort Duplication attacks in favor of keeping the regular Git `fetch` operation unchanged.
- Push certificates: Push certificates used upon pushing.

For our defense and our defense (light), the repositories were hosted on GitHub. Given that GitHub does not support push certificates, we studied the network overhead using a self-hosted server on an AWS instance, and concluded that push certificates incur a negligible overhead compared to the baseline. Thus, we only compare our scheme with push certificates in regard to the storage overhead.

Storage overhead. Table 6 shows the additional storage induced by our defense, compared to push certificates. In our defense, the RSL determines the size of the additional storage. We can see that our defense requires between 0.009%-6.5% of the repository size, whereas push certificates require between 0.012%-10%. The reason behind this is that push certificates contain 7 fields in addition to the signature, whereas RSL push entries only have 3 additional fields.

²Popularity is based on the “star” ranking used by GitHub, which reflects users’ level of interest in a project (retrieved on Feb 14, 2016).

Repo.	Our defense	Push certificates
R1	301.93	461.27
R2	6.49	8.88
R3	58.91	86.05
R4	44.34	66.27
R5	261.3	402.19

Table 6: Repository storage overhead of defense schemes (in KBs).

Communication overhead. To evaluate the additional network communication cost introduced by our `securepush` operation when compared to the regular push operation, we measured the cost of the last 10 pushes for the five considered repositories. To evaluate the cost of `securefetch`, we measured the cost of a fetch after each of the last 10 pushes.

Table 7 shows the cost incurred by push operations. We can see that our defense incurs, on average, between 25.24 and 26.21 KB more than a regular push, whereas our defense (light) only adds between 10.29 and 10.48 KB. This is because a `securepush` in our defense retrieves, updates and then stores the RSL in the remote repository. In contrast, our defense (light) only requires storing the RSL with the new push entry if there are no conflicts. Table 8 shows the cost incurred by fetch operations. A `securefetch` incurs on average between 25.1 and 25.55 KB more than a regular `fetch`, whereas our defense (light) only adds between 14.34 and 10.91 KB.

The observed overhead is a consequence of the fact that we implemented our defense scheme to respect design goal DG3, (*i.e.* no requirement to modify the Git server software). Since we implemented the RSL and the Nonce Bag as objects in a separate Git branch, `securepush` and `securefetch` require additional `push/fetch` commands to store/fetch these, and thus they incur additional TCP connections. Most of the communication overhead is caused by information that is automatically included by Git and is unrelated to our defense scheme. We found that Git adds to each `push` and `fetch` operation about 8-9 KBs of supported features and authentication parameters. If our defense is integrated into the Git software, the `securepush` and `securefetch` will only require one TCP session dramatically reducing the communication overhead. In fact, based on the size of an RSL entry (~325 bytes), which is the only additional information sent by a `securepush/securefetch` compared to a regular `push/fetch`, we estimate that the communication overhead of our defense will be less than 1KB per operation.

End-to-end delay. Table 9 shows the end-to-end delay incurred by push operations. We can see that our defense adds on average between 1.61 and 2.00 seconds more than a regular `push`, whereas our defense (light) only adds between 0.99 and 1.3 seconds. Table 10 shows

Scheme used	R1	R2	R3	R4	R5
Git w/ signed commits (baseline)	17.80	3,925.35	38.32	59.14	11.96
Our defense	44.01	3,950.87	63.56	84.71	37.65
Our defense (light)	28.28	3,935.71	48.61	69.52	22.28

Table 7: Average communication cost per push for the last 10 push operations, expressed in KBs.

Scheme used	R1	R2	R3	R4	R5
Git w/ signed commits (baseline)	20.68	3,896.98	40.93	65.85	13.67
Our defense	46.18	3,922.40	66.48	91.27	38.77
Our defense (light)	35.19	3,911.81	55.84	80.67	28.01

Table 8: Average communication cost per fetch for the last 10 fetch operations, expressed in KBs.

Scheme used	R1	R2	R3	R4	R5
Git w/ signed commits (baseline)	1.29	3.27	1.17	1.31	1.51
Our defense	3.11	5.27	2.78	2.95	3.51
Our defense (light)	2.44	4.49	2.16	2.40	2.81

Table 9: Average end-to-end delay per push for the last 10 push operations, expressed in seconds.

Scheme used	R1	R2	R3	R4	R5
Git w/ signed commits (baseline)	0.87	1.95	0.75	0.66	0.67
Our defense	2.93	3.86	2.52	2.40	2.75
Our defense (light)	1.60	2.75	1.52	1.31	1.30

Table 10: Average end-to-end delay per fetch for the last 10 fetch operations, expressed in seconds.

the end-to-end delay incurred by fetch operations. We can see that a `securefetch` incurs on average between 1.74 and 2.08 seconds more than a regular `fetch`, whereas our defense (light) only adds between 0.65 and 0.8 seconds.

The time Git uses to do a fetch or push is dominated by the network latency when talking with the remote repository. Since our defense is designed to be backwards compatible, it uses multiple Git commands per push or fetch. This explains the additional time incurred by our implementation. If our defense scheme is integrated into Git so that additional commands (and hence network connections) are not needed, we expect the additional delay to be negligible.

8 Conclusions

In this work, we present a new class of attacks against Git repositories. We show that, even when existing Git protection mechanisms such as Git commit signing, are used by developers, an attacker can still perform extremely impactful attacks, such as removing security patches, moving experimental features into production software, or causing a user to install a version of soft-

ware with known vulnerabilities.

To counter this new class of attacks, we devised a backwards compatible solution that prevents metadata manipulation attacks while not obstructing regular Git usage scenarios. Our evaluation shows that our solution incurs less than 1% storage overhead when applied to popular Git repositories, such as the five most popular repositories in GitHub.

We performed responsible disclosure of these issues to the Git community. We have been working with them to address these issues. Some of our patches have already been accepted into Git version 2.9. We are continuing to work with the Git community to fix these problems.

Acknowledgements

We would like to thank Junio C. Hamano, Jeff King, Eric Sunshine, and the rest of the Git community for their valuable feedback and insight regarding these attacks and their solutions as well as their guidance when exploring Git’s internals. Likewise, we thank Lois A. DeLong, Vladimir Diaz, and the anonymous reviewers for their feedback on the writing on this paper.

This research was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. A8650-15-C-7521, and by the National Science Foundation (NSF) under Grants No. CNS 1054754, DGE 1565478, and DUE 1241976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, AFRL, and NSF. The United States Government is authorized to reproduce and distribute reprints notwithstanding any copyright notice herein.

References

- [1] Apso: Secrecy for Version Control Systems. <http://aleph0.info/apso/>.
- [2] Git signed push. <http://thread.gmane.org/gmane.comp.version-control.git/255520>.
- [3] Github. <https://github.com>.
- [4] Kernel.org Linux repository rooted in hack attack. http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/.
- [5] 365 Git. Adding a GPG key to a repository. <http://365git.tumblr.com/post/2813251228/adding-a-gpg-public-key-to-a-repository>.
- [6] Ars Technica. “flame malware was signed by rogue ca certificate”. <http://arstechnica.com/security/2012/06/flame-malware-was-signed-by-rogue-microsoft-certificate/>.

- [7] Ars Technica. Lenovo pcs ship with man-in-the-middle adware that breaks https connections. <http://arstechnica.com/security/2015/02/lenovo-pcs-ship-with-man-in-the-middle-adware-that-breaks-https-connections/>.
- [8] Beta News. Has SSL become pointless? Researchers suspect state-sponsored CA forgery. <http://betanews.com/2010/03/25/has-ssl-become-pointless-researchers-suspect-state-sponsored-ca-forgery/>.
- [9] Briarproject. Development Workflow. <https://code.briarproject.org/akwizgran/briar/wikis/development-workflow>.
- [10] Bundler.io. Bundler: the best way to manage your application's GEMS. <http://bundler.io/git.html>.
- [11] Code in the hole. Using pip and requirements.txt to install from the head of a github branch. <http://codeinthehole.com/writing/using-pip-and-requirementstxt-to-install-from-the-head-of-a-github-branch/>.
- [12] Delicious Brains. Install wordpress site with Git. <https://deliciousbrains.com/install-wordpress-subdirectory-composer-git-submodule/>.
- [13] Extreme Tech. GitHub Hacked, millions of projects at risk of being modified or deleted. <http://www.extremetech.com/computing/120981-github-hacked-millions-of-projects-at-risk-of-being-modified-or-deleted>.
- [14] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proc. of the 9th USENIX Symposium on Operating Systems Design & Implementation (OSDI '10)*, 2010.
- [15] gamasutra. Cloud source host Code Spaces hacked, developers lose code. http://www.gamasutra.com/view/news/219462/Cloud_source_host_Code_Spaces_hacked_developers_lose_code.php.
- [16] Geek.com. Major Open Source Code Repository Hacked for months, says FSF. <http://www.geek.com/news/major-open-source-code-repository-hacked-for-months-says-fsf-551344/>.
- [17] M. Gerwitz. A Git Horror Story: Repository Integrity With Signed Commits. <http://mikegerwitz.com/papers/git-horror-story>.
- [18] Gigaom. Adobe source code breach; it's bad, real bad. <https://gigaom.com/2013/10/04/adobe-source-code-breach-its-bad-real-bad/>.
- [19] Git SCM. Signing your work. <https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work>.
- [20] M. Gunderloy. Easy Git External Dependency Management with Giteral. <http://www.rubyinside.com/giteral-easy-git-external-dependency-management-1322.html>.
- [21] E. Homakov. How I hacked GitHub again. <http://homakov.blogspot.com/2014/02/how-i-hacked-github-again.html>.
- [22] How To Node. Managing module dependencies. <http://howtonode.org/managing-module-dependencies>.
- [23] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing forged ssl certificates in the wild. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP '14*, pages 83–97, Washington, DC, USA, 2014. IEEE Computer Society.
- [24] I2P. Setting trust evaluation hooks. <https://geti2p.net/en/get-involved/guides/monotone#setting-up-trust-evaluation-hooks>.
- [25] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *Proc. of the 36th IEEE Symposium on Security and Privacy (S&P '15)*, 2015.
- [26] J. Li, M. Krohn, DMazieres, and D. Shasha. Secure untrusted data repository (sundr). In *Proc. of the 6th USENIX Symposium on Operating Systems Design & Implementation (OSDI '04)*, 2004.
- [27] LWN. Linux kernel backdoor attempt. <https://lwn.net/Articles/57135/>.
- [28] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, 2011.
- [29] J. Pellegrini. Secrecy in concurrent version control systems. In *Presented at the Brazilian Symposium on Information and Computer Security (SBSeg 2006)*, 2006.
- [30] RubyGems.org. Data verification. <http://blog.rubygems.org/2013/01/31/data-verification.html>.
- [31] Schneier on Security. Forging SSL Certificates. https://www.schneier.com/blog/archives/2008/12/forging_ssl_cer.html.
- [32] R. Shirey, K. Hopkinson, K. Stewart, D. Hodson, and B. Borghetti. Analysis of implementations to secure git for use as an encrypted distributed version control system. In *48th Hawaii International Conference on System Sciences (HICSS '15)*, pages 5310–5319, 2015.
- [33] Stack Overflow. How to install from specific branch with OPAM? <https://stackoverflow.com/questions/25277599/how-to-install-from-a-specific-git-branch-with-opam>.
- [34] Stack Overflow. Loading Maven dependencies from GitHub. <https://stackoverflow.com/questions/20161602/loading-maven-dependencies-from-github>.

- [35] Stack Overflow. Where does Cargo put the Git requirements? <https://stackoverflow.com/questions/28069678/where-does-cargo-put-the-git-requirements>.
- [36] The Art of Simplicity. TFS Build: Build from a tag. <http://bartwullems.blogspot.com/2014/01/tfs-build-build-from-git-tag.html>.
- [37] ThreatPost. Certificates spoofing google, facebook, godaddy could trick mobile users. <https://threatpost.com/certificates-spoofing-google-facebook-godaddy-could-trick-mobile-users/104259/>.
- [38] US-CERT. “SSL 3.0 Protocol Vulnerability and POODLE attack”. <http://arstechnica.com/security/2012/06/flame-malware-was-signed-by-rogue-microsoft-certificate/>.
- [39] D. A. Wheeler. Software Configuration Management (SCM) Security. <http://www.dwheeler.com/essays/scm-security.html>.
- [40] D. A. Wheeler. “The Apple goto fail vulnerability: lessons learned”. <http://www.dwheeler.com/essays/apple/goto-fail.html>.
- [41] Wired. Behind iphones critical security bug, a single bad goto. <http://www.wired.com/2014/02/gotofail/>.
- [42] Wired. ‘Google’ Hackers had ability to alter source code’. <https://www.wired.com/2010/03/source-code-hacks/>.
- [43] ZDNet. Gogo in-flight wi-fi serving spoofed ssl certificates. <http://www.zdnet.com/article/gogo-in-flight-wi-fi-serving-spoofed-ssl-certificates/>.
- [44] ZDNet. Open-source ProFTPD hacked, backdoor planted in source code. <http://www.zdnet.com/article/open-source-proftpd-hacked-backdoor-planted-in-source-code/#!>
- [45] ZDNet. Red Hat’s Ceph and Inktank code repositories were cracked. <http://www.zdnet.com/article/red-hats-ceph-and-inktank-code-repositories-were-cracked/#!>

A Man In The Middle Example

This appendix contains a proof of concept of a Git metadata manipulation attack against a GitHub repository with the intention of showing how an attack could be carried out in practice.

To perform an attack of this nature, an attacker controls a server, compromises a server, or acts as a man-in-the-middle between a server and a developer. Having done this, the attacker is able to provide erroneous metadata to trick a developer into committing a tampered repository state.

We simulated a repeated line scenario, in which a Git merge accidentally results a repeated line. This can be devastating as it can completely alter the flow of a program — some researchers argue that the “goto fail;” [41] vulnerability that affected Apple devices [40] might have been caused by a VCS mistakenly repeating the line while merging.

A.1 Simulating the attack

To simulate the attack, we created a repository with a minimal working sample that resembles Figure 5(c). Also, we configured two Linux machines under the same network: one functioned as the malicious server providing tampered metadata information, while the other played the role of the victim’s client machine. The specific setup is described below.

Setup. To simulate the malicious server, we set up Git server on port 443 with no authentication enabled. Then, we created an SSL certificate and installed it in the victim machine. Finally, we a bare clone (using the `--bare` parameter) of the repository hosted on GitHub is created and placed on the pertinent path.

In order to redirect the user to the new branch, we modified the packed-refs file on the root of the repository so that the commit hash in the master branch matches the one for the experimental branch. Refer to Table 11 for an example.

On the client side, a clone of the repository is created before redirecting the traffic. After cloning, the attacker’s IP address is added to the victim’s `/etc/hosts` file as “github.com” to redirect the traffic.

As such, both the server and the developer are configured to instigate the attack the next time the developer pulls.

A.2 The attack

When the developer pulls, he or she is required to either merge or rebase the vulnerable changes into the working branch. These merged or rebased changes are not easy to identify as malicious activity, as they just resemble work performed by another developer on the

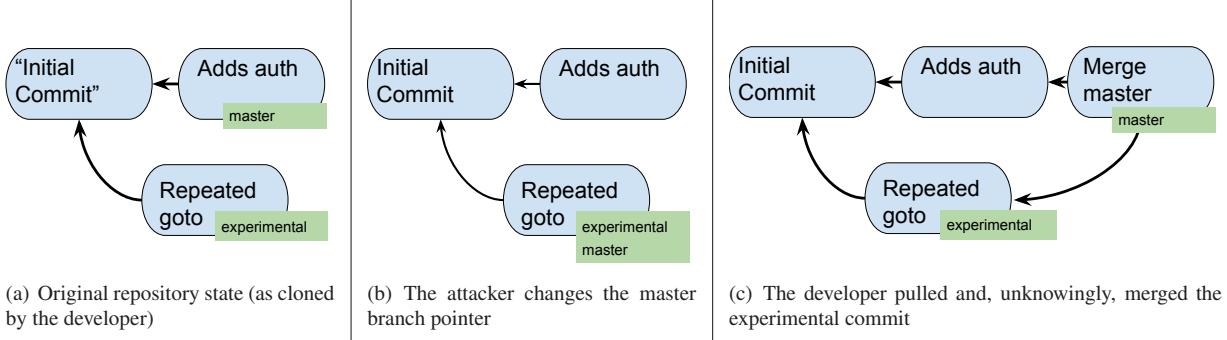


Figure 5: Maliciously merging vulnerable code

Original file
pack-refs with: peeled fully-peeled
00a5c1c2f52c25fe389558ea8117b7914ca2351e refs/heads/experimental
3a1db2295a5f842d0223088447bc7b005df86066 refs/heads/master
Tampered file
pack-refs with: peeled fully-peeled
00a5c1c2f52c25fe389558ea8117b7914ca2351e refs/heads/experimental
00a5c1c2f52c25fe389558ea8117b7914ca2351e refs/heads/master

Table 11: The edited packed-refs file

same branch. Due to this, the user is likely to merge and sign the resulting merge commit.

Aftermath. Once the user successfully merges the vulnerable change, the attacker can stop re-routing the user’s traffic to the malicious server. With the malicious piece of code in the local repository, the developer is now expected to pollute the legitimate server the next time he or she pushes. In this case, the attacker **was able to merge a vulnerable piece of code into production**. Even worse, there is no trace of this happening, for the target developer willingly signed the merge commit object.

Setting up an environment for this attack is straightforward; the metadata modification is easy to perform with a text editor and requires no sophistication.

Defending against malicious peripherals with Cinch

Sebastian Angel,^{*†} Riad S. Wahby,[‡] Max Howald,^{§†} Joshua B. Leners,^{||}
Michael Spilo,[†] Zhen Sun,[†] Andrew J. Blumberg,^{*} and Michael Walfish[†]

^{*}The University of Texas at Austin [†]New York University [‡]Stanford University [§]The Cooper Union ^{||}Two Sigma

Abstract

Malicious peripherals designed to attack their host computers are a growing problem. Inexpensive and powerful peripherals that attach to plug-and-play buses have made such attacks easy to mount. Making matters worse, commodity operating systems lack coherent defenses, and users are often unaware of the scope of the problem. We present Cinch, a pragmatic response to this threat. Cinch uses virtualization to attach peripheral devices to a logically separate, untrusted machine, and includes an interposition layer between the untrusted machine and the protected one. This layer regulates interaction with devices according to user-configured policies. Cinch integrates with existing OSes, enforces policies that thwart real-world attacks, and has low overhead.

1 Introduction

Peripheral devices are now powerful, portable, and plentiful. For example, the inexpensive “conference USB sticks” that we have all received include not only the stored conference proceedings but also a complete computer. Given this trend, it is easy to create *malicious* peripheral devices [43, 61, 88, 98]. And yet, it is difficult to defend against them: commodity machines and operating systems continue to be designed to trust connected peripherals.

Consider a user who is induced to insert a malicious USB stick into his or her laptop [91, 135, 148]. There are now many examples [16, 75, 89] of such devices injecting malware (most infamously, Stuxnet [94]), by exploiting vulnerabilities in the host’s drivers or system software.

Another alarming possibility is that, while following the USB specifications, the malicious device can masquerade as a keyboard. The device can then use its keystroke-producing ability to install a virus or exfiltrate files [43, 61, 125, 150]. As a last example, a USB device can eavesdrop on the communication between another device, such as the user’s true keyboard, and the host [12, 17, 25, 72, 124].

These problems will get worse: on next-generation laptops [5, 10], *all* ports, including the power port, are USB, which means that any of the attacks above could be carried out by a malicious charger. For that matter, your phone might be compromised right now, if you borrowed a USB charger from the wrong person.

On the one hand, the concepts needed to solve these problems have long been understood. For example, in

Rushby’s separation kernel [129] (see also its modern descendants [81, 122]), the operating system is architected to make different resources of the computer interact with each other as if they were members of a distributed system. More generally, the rich literature on high-assurance kernels offers isolation, confinement, access control, and many other relevant ideas. On the other hand, applying these works in full requires redesigning the operating system and possibly also the hardware.

Solutions that target device security for today’s commodity systems are not adequate for the task, often because they were designed under different models (§8). For example, work on device driver containment [80, 83, 93, 95, 96, 105, 112, 114, 127, 143–145, 152] and reliability [108, 130–132] trusts devices or assumes they are at worst buggy; the attacks mentioned earlier are largely out of scope. Hotplug control frameworks [13, 15, 18, 22, 33, 35, 37, 48, 50, 55], of which a notable example is udev on Linux [56, 110], enable users to express that certain devices should be denied access. However, access is all-or-nothing, decisions are based upon the device’s claimed identity rather than its ongoing behavior, and a malicious device can disarm the enforcement mechanism. Qubes [45] protects the OS and applications from malicious USB devices, but achieves its strong guarantees at the expense of functionality.

The fundamental issue is that the I/O subsystems in commodity operating systems do not have an organizing abstraction that could serve as a natural foundation for security features. This paper attempts to fill that void.

Our point of departure is a simple suggestion: rather than design a new framework, *why not arrange for attached peripheral devices on commodity operating systems to appear to the kernel as if they were untrusted network endpoints?* This would create an interposition point that would allow users and administrators to defend the rest of the computer, just as firewalls and other network middleboxes defend hosts from untrusted remote hosts. Our animating hope is that a system based on this picture would eliminate large classes of vulnerabilities, be easy to deploy, and enable new functionality. To explore that vision, this paper describes the design, implementation, and experimental evaluation of a system called *Cinch*. Cinch begins with the following requirements:

- *Cinch should make peripheral buses look “remote,” despite the physical coupling*, by preventing direct inter-

action with the rest of the computer (memory access, interrupts, etc.).

- *Under Cinch, traffic between the “remote” devices and the rest of the computer should travel through a narrow choke point.* This choke point then becomes a convenient location for deploying defenses that inspect and mediate interactions with untrusted devices.
- *Cinch should not require modifying bus standards, motherboards, OSes, or driver stacks.* Any of these would be massive undertakings, would have to be done for multiple platforms, and would jettison the immense effort behind today’s installed base.
- *Cinch should be portable,* in the sense that Cinch itself should not need to be re-designed or re-implemented for different operating systems.
- *Cinch should be flexible and extensible:* users, operators, and administrators should be able to quickly develop and deploy a wide range of defenses.
- *Cinch should impose reasonable overhead* in latency and throughput.

Cinch responds to these requirements with the following architecture, focused on USB as a target (§4). Under Cinch, USB devices attach to an isolated and untrusted module; this is enforced via hardware support for virtualizing I/O [70, 71]. The untrusted module tunnels USB traffic to the protected machine, and this tunnel serves as a choke point for enforcing policy.

To showcase the architecture, we build several example defenses (§5). These include detecting attacks by matching against a database of attack signatures (§5.1); sanitizing inputs by ensuring that messages and device state transitions comply with protocol and device specifications (§5.2); sandboxing device functions and enforcing hotplug policies (§5.3); device authentication and traffic encryption (§5.4); and logging and remote auditing (§5.5).

Our implementation of Cinch (§6) instantiates both the untrusted module and the protected machine as separate virtual machines. As a consequence, Cinch protects any OS that runs atop the underlying hypervisor. In principle, these virtualization layers can be reduced or eliminated, at the cost of development effort and portability (§4.2).

To study Cinch’s effectiveness, we developed exploits based on existing vulnerabilities [14], performed fuzzing, and conducted an exercise with a red team whose members were kept isolated from Cinch’s development (§7.1–§7.3). Our conclusion is that Cinch can prevent many attacks with relatively little operator intervention. We also find that developing new defenses on Cinch is convenient (§7.4). Finally, Cinch’s impact on performance is modest (§7.5): Cinch adds less than 3 milliseconds of latency and can handle USB 3 transfers of up to 2.1 Gbps, which is 38% less than the baseline of 3.4 Gbps.

Cinch is enabled—and inspired—by much prior work

in peripherals management, hardware-assisted virtualization, privilege separation, and network security. We delve into this work in Section 8. For now, we simply state that although Cinch’s individual elements are mostly borrowed, it is a novel synthesis. That is, its contributions are not mechanical but architectural. These contributions are: viewing peripherals as remote untrusted endpoints, and the architecture that results from this perspective; the instantiation of that architecture, which uses virtualization techniques to target a natural choke point in device driver stacks; a platform that allows defenses to existing attacks to be deployed naturally on commodity hardware, in contrast to the status quo; and the implementation and evaluation of Cinch.

Cinch is not perfect. First, it shrinks the attack surface that the protected machine exposes to devices, but introduces new trusted code elsewhere (§4.2). Second, although Cinch can reduce the universe of possible inputs to the drivers and OS on the protected machine (by ruling out noncompliant traffic), a malicious device might still exploit bugs in how the code handles *compliant* traffic. On the other hand, the user can decide which devices get this opportunity; further, addressing buggy drivers and system software is a complementary effort (§8). Third, Cinch does not unilaterally defend against higher-level threats (data exfiltration, malware, etc.); however, Cinch creates a platform by which one can borrow and deploy known responses from network security (§5). Finally, some of Cinch’s defenses require changes within the device ecosystem (§9). For example, defending against masquerading attacks requires device (but not bus) modifications. However, these changes are limited: in our implementation, one person prototyped them in less than two days (§6.3). Importantly, these changes can be used with unmodified legacy devices via an inexpensive adapter.

Despite its shortcomings, Cinch is a substantial improvement over the status quo when considering the misbehavior that it rules out and the functionality that it enables. Moreover, we hope that Cinch’s perspective on device security will be useful in its own right.

2 Background: Universal Serial Bus (USB)

Commodity computing devices (phones, tablets, laptops, workstations, etc.) have several peripheral buses for pluggable devices. These include USB [57, 58], Firewire [1], and Thunderbolt [54]. Cinch focuses on USB as an initial target; we make this choice because USB is ubiquitous and complex, and because it has become a popular locus of hardware-based attacks. However, our approach applies to other buses.

Figure 1 depicts the hardware and software architecture of USB. USB is a family of specifications for connecting and powering peripheral devices. Bandwidth ranges from 1.5 Mb/s (USB 1.0) to 10 Gb/s (USB 3.1). Example

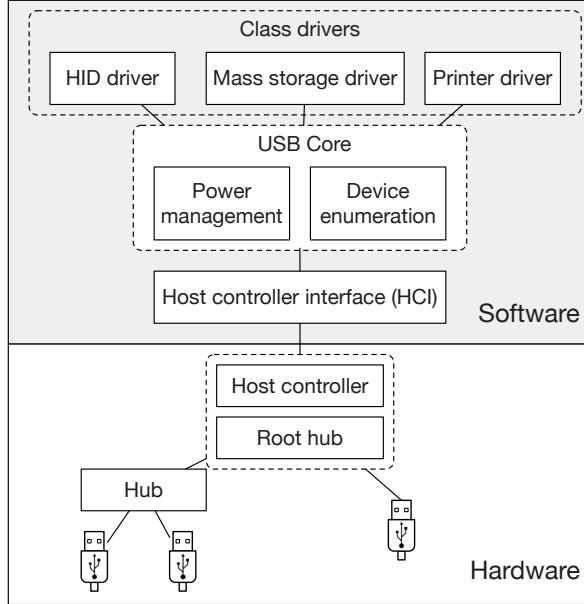


FIGURE 1—The hardware and software of a USB stack (§2). Both physical devices and drivers are arranged hierarchically; devices are rooted at the host controller, and drivers are rooted at the host controller interface. Components in dashed boxes are logically in the same layer of the USB stack.

devices include storage (e.g., memory sticks), keyboards, sound cards, video cameras, Ethernet adapters, and smart card readers. These devices connect to a host (for example, a laptop or desktop). Some computers can act as either a device or a host; for example, a smart phone or laptop can appear as a storage device or power consumer to a desktop, but as a host to a keyboard.

USB hardware. USB has a tree topology. Each device has an *upstream* connection to a *hub*. Hubs multiplex communication from one or more *downstream* devices, and are themselves devices with an upstream connection to another hub or to the root of the tree. The root is a *host controller*, which connects to the host by, for example, PCIe. The host controller acts as the bus master: it initiates all transfers to and from devices, and devices are not permitted to transmit except when polled by the host controller. Also, the host controller issues interrupts to the host and has direct access to host memory via DMA.

USB protocol. The USB specifications [57, 58] define a protocol stack comprising three layers. The bottom layer includes electrical specifications and a low-level packet protocol. The middle layer of the stack includes addressing, power management primitives, and high-level communication abstractions. USB devices, comprising one or more *functions*, sit at the top of the stack. Functions act as logically separate peripherals that are exposed by a single physical device. For example, a phone might ex-

pose a camera function, a network adapter function, and a storage function. Each of these functions is associated with its own high-level driver software.

USB driver architecture. The USB specification describes three layers of software abstraction on the host. The lowest level, the *host controller interface* or *HCI*, configures and interacts with the host controller hardware via a local bus (e.g., PCIe). An HCI driver is particular to a host controller’s hardware interface but exposes a hardware-independent abstraction to the next software layer, called *core*. Core manages device addressing and power management, and exposes an interface for high-level drivers to communicate with devices. Core also *enumerates* devices when they are attached, which entails identifying the device and activating its driver.

The uppermost layer, *class drivers*, are high-level drivers that interact with functions (as described above). These drivers provide an interface between USB devices and the rest of the OS. For example, a keyboard’s class driver interacts with the kernel’s input subsystem. Another example is the mass storage class driver, which talks to the kernel’s storage subsystem. The USB specification defines a set of generic classes for a broad range of devices, e.g., keyboards, mice, network interfaces, storage, cameras, audio, and more. Operating systems generally include support for a large subset of the generic classes, allowing devices to leverage preexisting drivers.

3 Causes, threat model, and taxonomy

3.1 Why is USB so vulnerable?

The root of the problem is the implicit assumption that hardware is inherently trustworthy, or at worst buggy but non-malicious. As a consequence, neither USB nor mainstream OSes are designed to be robust in the face of malicious devices. One manifestation of this is the lack of authentication or confidentiality guarantees at any layer of the USB standard. As examples, devices self-report their identity and capabilities without authentication; the communication primitives at all layers of the protocol stack (§2) are cleartext; and, prior to USB 3, host-to-device messages are broadcast across the entire bus [124].

A related issue is that the USB protocol and common driver stacks emphasize convenience above correctness and security. For example, hotplugged devices are often activated without user confirmation. Coupled with the lack of device authentication, this means that the OS cannot determine what device the user intended to connect, or even that a hotplug event was generated by the *user* rather than a malicious device [24, 75]. Moreover, malicious device makers can rely on the near universal availability of generic class drivers (e.g., for keyboards), since users expect these devices to “just work.”

The range and sophistication of USB-based threats has escalated substantially in recent years. Whereas hardware design costs were once a barrier to entry, creating custom USB devices is now cheap, both in dollars and development time [43, 52, 61, 98, 100]; in fact, today’s commodity USB devices are essentially software defined [43, 75, 98].

The press plays a role too: demonstrating USB attacks has become fashionable (e.g., recent media hype [6–8, 39, 53, 118] surrounding USB devices with reprogrammable firmware [43, 84, 125]). A third factor is ease of transmission: malicious USB devices can easily find their way into the hands of victims [148]. This is partly due to vulnerabilities in the supply chain [38, 74, 101, 141], such as adversarial manufacturers [102]. Intelligence agencies have also been known to use their resources to intercept and “enhance” shipments [27, 97], including conference giveaways [20, 21].

3.2 Threat model

We assume that devices can deviate from the USB specification arbitrarily. They may also violate the user’s expectations, for example by masquerading as other devices or passively intercepting bus traffic. Alternatively, devices can present a higher-level threat; for example, a storage device can contain an invalid filesystem that triggers a bug in a filesystem driver. However, devices that cause physical damage to the host, with high voltage [86] for example, are out of scope.

We assume that the host’s OS and drivers can be buggy but not malicious. We assume the same for the host’s hardware besides the USB controller and USB devices.

3.3 A taxonomy of USB attacks

Attacks on USB drivers. USB drivers present an attack surface to devices. For example, a driver with an unchecked buffer access might allow a malicious device to overwrite kernel memory via an overflow. The space of possible misbehavior here is vast. For instance, devices might try to deliver more data to the driver than indicated by the device’s configuration [31]; claim impossible configurations [28, 30]; exceed limits prescribed by USB class specifications [4, 32, 42]; or produce otherwise invalid or nonsensical reports [75, 87–89, 92, 137].

The prevalence of these attacks reflects a difficult software engineering situation. Since a driver writer needs to be prepared for an enormous range of undocumented behavior, drivers need lots of error checking code; such code is often ill-exercised and creates complexity, leading to more vulnerabilities. Indeed, more than half of the vulnerabilities related to USB drivers in the CVE database [14] are the result of improper handling of noncompliant USB transfers; many more such vulnerabilities likely remain undisclosed [87, 137].

Other attacks on the host via USB. USB can also expose the rest of the host system’s kernel or user software to attacks by malicious devices. Recall that USB class drivers provide an interface between devices and other kernel subsystems (§2). Leveraging this interface, a USB flash drive might be used to attack the kernel’s storage or filesystem drivers [19, 44, 63, 64]. Or the drive might carry a virus [94] or covertly steal data [140].

Of particular concern is the possibility of attacks in which the USB host controller uses DMA (direct memory access) to bypass the CPU and read or write arbitrarily to RAM [26, 116, 139, 142]. A successful DMA attack neutralizes essentially all software security measures, allowing the attacker to steal sensitive data, modify running software (including the kernel itself), and execute arbitrary code [128]. And the host controller does not need to be malicious: misconfigured DMA-capable hardware is a proven vector for such attacks [153, 154].

Privacy and authentication threats.

Device masquerading. When a device is plugged in, the host asks the device for information about its capabilities. The device can respond, disguised as another device or even another class [29, 41, 43, 65, 85, 120, 125, 150]. For example, Psychson [43] enables rewriting the firmware on a cheap USB storage device so that it will act like a keyboard; similarly, the commercially available “USB Rubber Ducky” [61] is a programmable keystroke injector in the guise of a flash drive. Likewise, a malicious hub can masquerade as other devices [25]. These examples are more than idle threats: penetration testers regularly use such tools to breach security systems [3, 24].

Bus eavesdropping. In USB 2 and earlier versions, hubs broadcast traffic from their upstream port to all downstream ports (§2), so any device on the bus can eavesdrop on traffic from the host to any other device [124]. In all protocol versions, malicious hubs can eavesdrop on upstream and downstream traffic [17, 72]. Furthermore, a hub need not be malicious: if its firmware is buggy, it can be exploited by a malicious device [25].

4 Architecture and rationale

The top-level goal of Cinch is to enforce security policies that enable safe interactions between devices and the host machine. This enforcement must be done in a way that respects the requirements outlined in Section 1. In particular, we must answer two questions in the context of USB: (1) Where and how can one create a logical separation between the bus and the host, while arranging for an explicit communication channel that a policy enforcement mechanism can interpose on? (2) How can one instantiate this separation and channel with no modifications to bus standards, OSes, or driver stacks?

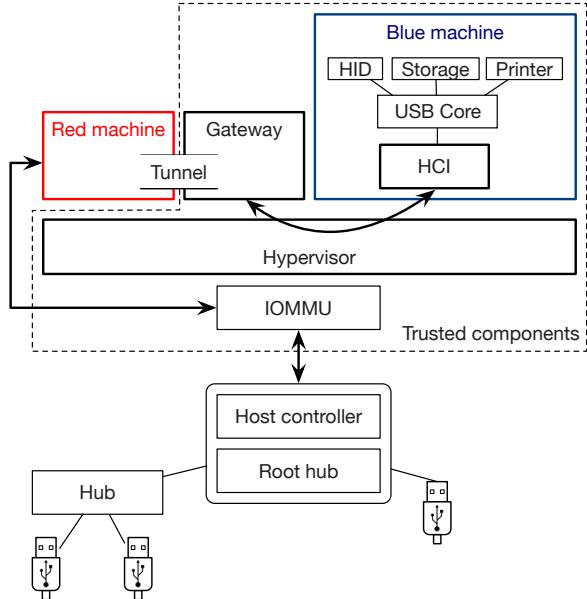


FIGURE 2—The architecture of Cinch. The trusted components are surrounded by the dashed line. I/O virtualization separates the USB host controller from the blue machine’s HCI, redirecting DMA and interrupts to the red machine. The red machine encapsulates and sends USB transfers through the Tunnel to the Gateway. Once the Gateway has applied all security policies, it redirects those transfers to the blue machine’s HCI.

We begin with the logical separation, which Cinch enforces at the boundary between the host controller and its driver (HCI), depicted in Figure 1. We choose this separation point for two reasons: first, it results in a narrow choke point where software can interpose. Second, the host controller is “dangerous”—it issues interrupts and accesses memory via DMA (§2, §3.3)—so there should be a barrier between it and the rest of the system, including the modules that administer policy decisions.

The architecture is depicted in Figure 2. After logically separating the host controller, Cinch attaches it to a new module, the *red machine*. The red machine is an endpoint to a communication channel, the *Tunnel*. The other endpoint, the *Gateway*, is positioned at the entrance to the host that Cinch protects, the *blue machine*. (These names are inspired by Lampson’s red/green machine partitioning [111].) The Gateway mediates all transfers through the Tunnel and enforces security policies (for example, dropping or rewriting USB traffic, as described in §5) before those transfers reach the blue machine’s USB stack.

To connect the host controller to the red machine, Cinch uses I/O virtualization hardware, which is widely available in modern CPUs [70, 71]. Specifically, an IOMMU provides address translation and protection, which restricts a physical device’s DMA transfers to a designated memory region (in this case, that of the red machine); and interrupt remapping provides analogous translation and protection for interrupts.

4.1 Instantiation

In our current implementation, the lowest layer of software—the one that manages the hardware resources and configures the I/O virtualization hardware—is a combination of hypervisor and OS, and is trusted. The red machine runs on top of this hypervisor and is a full-fledged virtual machine, with a normal OS that has a stripped-down USB stack (§6.1). The blue machine is also a full-fledged virtual machine atop the hypervisor, and the Gateway is a separate process.

4.2 Discussion

With the instantiation described immediately above, Cinch meets the requirements described in Section 1. It isolates devices in the red machine, and its Gateway is a narrow choke point. It limits overhead to reasonable factors (§7.5), in part by leveraging hardware-assisted processor and memory virtualization [68, 123] (as distinct from I/O virtualization). It works with existing USB stacks; the main component needed is a driver in the hypervisor, to receive transfers from the Gateway. It works with a range of OSes because the blue machine runs unmodified. For the remaining requirements, flexibility is demonstrated in the next section (§5), and extensibility arises from Cinch’s software structure (§6.2).

But a disadvantage is the size of the trusted computing base (TCB) and attack surfaces. Specifically, the TCB includes a full-featured hypervisor. The attack surface includes the red machine, which is running a full OS and which, if compromised, can attack the hypervisor and the blue machine via the virtualization interface (by attempting VM escapes, side channel inference, etc.).

There are a number of alternatives that, by tailoring the hypervisor and red machine, reduce the TCB at the cost of portability and additional development effort. As an extreme example, the blue machine could run directly on the host’s hardware (“bare metal”), with the red machine running in an untrusted user-level process; the Gateway would also run in user space. In this setup, there would be no separate hypervisor; the blue machine would perform the few required hypervisor-like functions, such as configuring the I/O virtualization hardware to connect the host controller to the red machine process (see [77, 78, 126]). Compared to Cinch’s instantiation, this one has a smaller TCB; it also has lower overhead, owing to the absence of virtual machines. However, it is less portable: each new blue machine OS needs corresponding “hypervisor” module and red machine implementations.

One can go further: the Gateway could entirely bypass the blue machine’s USB stack, sending device traffic directly to the corresponding kernel subsystem (for example, sending USB keyboard events to the input subsystem). This would further reduce the TCB, at the cost of even more development work and less portability.

Another design point is a hardware-only solution: the red machine and Gateway would run on a device placed between USB devices and the blue machine, which would run as a normal, unmodified host. Compared to Cinch, this solution is potentially more portable, in that no software modifications or reconfiguration are needed. Further, this solution does not rely on I/O virtualization (which is widespread but not universal), and it leaves the host’s virtualization hardware available for other uses. The disadvantages are that a hardware solution is likely to be less flexible, and that building hardware may be substantially more effort than building Cinch.

A non-solution, in our view, is to implement the Gateway in the host’s USB stack, without a separate red machine. This setup does not have the separation discussed earlier; it would leave the host and Gateway vulnerable to DMA attacks by a compromised host controller.

5 Building defenses with Cinch

This section describes some of the defenses (which we call *Policies*) that Cinch supports, and the threats (§3) against which they defend. These Policies are not new; we discuss previous implementations in Section 8. The novelty is in providing a platform that makes a range of Policies straightforward to develop and deploy.

5.1 Detecting attacks by signature

The first strategy is signature matching: dropping messages that match a known pattern. Defenses in this class protect against attacks on drivers and user software (§3.3). The same strategy is used in network security (intrusion detection [47]) and desktop security (antivirus [11]) and has been effective in practice, as a first-line defense. The advantages and disadvantages hold in our context; we review them briefly.

To begin with, signature generation is flexible and can be done by victimized companies, individual users, and designated experts, based on observations of past attacks and reverse engineering of malicious devices. Further, shared databases of observed attack signatures can immunize others. This strategy also enables rapid responses to emerging threats: a signature of an attack is typically available long before the vulnerability is patched.

The principal disadvantage, of course, is that signatures generally provide protection only against previously observed attacks. Furthermore, they suffer from both false positives and false negatives: signatures that are too general may disable benign devices, while signatures that are too specialized can fail to catch all variants of an attack.

Cinch’s signature Policy. We implement a signature matching module in Cinch that compares all USB traffic from the red machine to a database of malicious payload signatures. When a match occurs, Cinch disallows further traffic between the offending device and the blue machine.

5.2 Sanitizing inputs

Another class of defensive strategies detects when devices deviate from their specification; this is useful for defending against attacks on USB drivers (§3.3). Given a specification (say, provided by the manufacturer or converted from a standards document), Cinch checks that messages are properly formatted and that devices respond correctly to commands. While drivers can (and in some cases, do) implement such checks, moving enforcement to a dedicated module can eliminate redundant code and reduce driver complexity (§3.3, “Attacks on USB drivers”).

A related strategy in Cinch modifies apparent device behavior, either forcing adherence to a *strict subset* of the USB spec in order to match driver expectations, or else *relaxing* the USB spec by recognizing and fixing device “quirks”—behavior that is noncompliant but known to be benign—so that drivers need not do so.¹ This is closely related to traffic normalization [103], in which a firewall converts traffic to a canonical representation to aid analysis and ensure that decisions are consistent with end-host protocol implementations.

Cinch’s compliance Policy. This Policy enforces device compliance with USB specifications. To build it, we manually processed the USB 2 and 3 specifications [57, 58], along with the specifications of five device classes (mass storage, HID, printer, power, and debug) [59]. The result is a module that monitors device states and transitions, and enforces invariants on individual messages and entire transactions. As a simple example, the compliance Policy checks that device-supplied identification strings are well formed—that is, that they comprise a valid UTF-16 string of acceptable length—and rewrites noncompliant strings. More complex state and transition checking is effected by keeping persistent information about each device for the duration of its connection.

Cinch’s compliance Policy is conservative in handling noncompliance: if it cannot easily fix a device’s behavior (for example, by rewriting identification strings as described above), it assumes the device is malicious and disables it.

Cinch’s assertion Policy. This Policy implements the aforementioned relaxations and restrictions, by modifying how Cinch’s compliance Policy regulates specific devices. As examples, a user might specify that a particular device’s requests should be rewritten to work around buggy firmware. Or Cinch can require that devices handled by a certain driver must expose an interface that matches a specified template, obviating bug-prone compatibility checks in the driver’s code (§3.3).

¹In practice, many non-malicious devices fail to comply with the specification: the word “quirk” appears about once every 300 lines throughout the 300 kLoC Linux USB stack (!), and nearly all the devices we tested deviated from prescribed behavior in at least a small way.

5.3 Containing devices

This category includes querying a user for information about a newly connected device, restricting a device to a subset of its functionality, and isolating devices in private protection domains. Such defenses, which are useful against attacks on driver and user software and can foil masquerading attacks (§3.3), are forms of *hotplug control*. They decide—say, by asking the user—whether a newly connected device should be allowed to communicate with the blue machine, and if so, what functionality should be allowed. For example, Cinch might ask the user, “I see you just connected a keyboard. Is this right?”

In practice, such decisions can be much more complex. Recall from Section 2 that devices can define multiple functions, each of which is a logically separate peripheral. A careful user wishing to tether his or her laptop to a friend’s phone could be informed of available functionality upon device connection, and choose to disallow the phone’s storage function as a precaution against viruses.

Alternatively, the user might choose to connect the phone’s file storage function to a separate protection domain—a sandbox—with limited capabilities and a narrow interface to the blue machine. In this case, the sandbox could scan files for viruses, and could expose a high-level interface (e.g., an HTTP or NFS server) to the blue machine. This approach leverages existing software designed for interacting with untrusted machines (in this case, a web or file browser), and can bypass many layers of software in the blue machine; on the other hand, it changes the interface to the device.

Cinch’s containment Policy. We implement a “surgical” hotplug Policy: individual device functions can be allowed or disallowed, and the blue machine never interacts with disallowed devices. Cinch’s Gateway can also sandbox whole devices or individual functions by redirecting selected USB traffic to separate protection domains that expose functionality to the blue machine through narrow interfaces, as described above.

5.4 Encryption and authentication

To handle devices that eavesdrop on the bus or masquerade as other devices, Cinch adapts well-known responses—authentication and encryption—to USB. For example, a user can disallow all keyboards except those having a certificate signed by a particular manufacturer. This prevents a malicious device without such a certificate from acting as a keyboard.

In more detail, a device authenticates to the Gateway by leveraging a trust relationship. As examples, manufacturers sign certificates and install them on devices, and users are required to use devices whose certificates are signed by a trusted manufacturer; or users follow a pairing procedure as in Bluetooth [67] or GoodUSB [146], obvi-

ating a trusted manufacturer but adding a setup step. After completing a key exchange, the device and host share an encryption key. The user can then prevent masquerading and eavesdropping by installing a policy that disallows unauthenticated, untrusted, or unencrypted devices.

This arrangement raises several potential concerns: development overhead to build new devices, computational overhead for cryptography, and deployment on legacy devices. Below, we describe a proof-of-concept design that addresses these concerns. At a high level, the concerns are addressed by, respectively, abundant support for rapid development of embedded cryptographic applications [34, 36, 66], the speed of modern embedded processors, and a physical *adapter* that adds cryptographic functionality to legacy devices.

Proof-of-concept USB crypto support. To support authentication and encryption, we designed a *cryptographic overlay protocol*. This mechanism allows compatible devices to communicate with the Gateway via a TLS session that encapsulates all of their USB transfers.

To evaluate the crypto overlay, we built a *crypto adapter*, a physical device that sits between unmodified legacy devices and a host system running Cinch. The crypto adapter acts as a USB host for the legacy device, encapsulating and decapsulating the device’s USB traffic inside a TLS session. To communicate this TLS-encrypted traffic to the host system, the crypto adapter also acts as a USB device attached to the host system, as we detail below. We refer to the crypto adapter’s USB connection to the legacy device as the “inner” connection, and its connection to the host as the “outer” connection.

Two issues arise in designing the crypto overlay and adapter. First, a TLS session requires a full duplex stream transport, while USB’s communication primitives are based on host-initiated polling (§2). This means that the outer USB connection cannot directly encapsulate a TLS session. Second, the Gateway does not implement a USB stack, meaning that, on its own, it cannot communicate with the crypto adapter via the outer USB connection.

To solve the first issue, Cinch uses an existing USB class that exposes a full-duplex Ethernet interface [59]; this Ethernet-over-USB traffic is carried by the outer USB connection. Then Cinch uses TCP over this Ethernet connection as the stream abstraction for TLS, yielding an indirect encapsulation of TLS in the outer USB connection.² To solve the second issue, we observe that, with the foregoing encapsulation, the Gateway need not handle the outer USB connection. Instead, the red machine treats the outer USB connection as an Ethernet device (thereby terminating the outer USB connection), and it forwards all packets it receives from that device to the Gateway via

²An alternate approach with less overhead than TCP-over-IP-over-Ethernet-over-USB is to create a custom USB class providing a full-duplex stream abstraction with less generality than Ethernet.

the Tunnel. Meanwhile, these packets are just the TCP stream carrying the TLS session, and thus the Gateway can talk TLS to the crypto adapter without a USB stack.

Note that this arrangement differs from the way that Cinch handles other USB devices. For unencrypted devices, the Gateway receives USB transfers captured by the red machine; it inspects these transfers and then forwards them to the blue machine’s HCI. But here, the Gateway receives packets (which the red machine decapsulated) that contain a TLS session. The Gateway decrypts to recover USB transfers, which it inspects and forwards.

Cinch’s crypto Policy. Given devices implementing the crypto overlay, Cinch can enforce policies that rule out eavesdropping and masquerading by requiring authenticated and encrypted devices, as described at the outset of this section.

5.5 Logging and auditing

Logging is part of many defensive strategies: auditing logs can reveal anomalous behavior that might indicate a new attack. Moreover, logs can be used to develop new signature-based defenses (§5.1).

Cinch’s logging Policy. Cinch’s Gateway can be configured to log some or all traffic to and from the blue machine. Cinch can also replay logged data; we used this functionality to help develop attack signatures for our security evaluation (§7.3). Furthermore, Cinch can be configured to log to a remote server. This feature could allow real-time analysis of data from many different blue machines, for example in a corporate environment.

5.6 Extensions

Cinch enables usage scenarios beyond the ones described above. One example is data exfiltration prevention, which is often employed at the network level to address the threat of data theft [104, 115, 117, 133, 134], but is generally considered a more difficult problem in the context of USB [140]. By combining real-time remote auditing (§5.5) with signature detection (§5.1), Cinch allows administrators to apply exfiltration prevention policies to USB devices.

6 Implementation

We describe the components and the communication paths in our implementation of Cinch (§6.1). We also discuss the Policies implemented in Cinch, utilities that we use to create and test new exploits, and our method for deriving payload signatures (§6.2). Finally, we describe the proof-of-concept crypto adapter (§5.4) that we use to transparently provide encryption and authentication for existing USB devices (§6.3).

6.1 Components and communication paths

The hypervisor (§4.1) is Linux with KVM, meaning that virtual machines run in QEMU processes that are accelerated with virtualization hardware [68, 123]. In particular, Cinch requires hardware support for I/O virtualization [70, 71]. We tested with Intel hardware, but KVM also supports equivalent functionality from AMD.

The red machine runs Linux. It is configured to load only the HCI and core drivers (§2); higher-level USB drivers are not needed to capture USB transfers from devices. (An exception is the case of the crypto overlay, which requires a USB network driver; §5.4). The blue machine is another VM and, as stated in Section 4.2, can be any OS supported by QEMU. The Gateway runs as a user-level process on the Linux-KVM hypervisor.

The Tunnel between the red machine and the Gateway appears to both entities as a network device. The appeal of this approach is that the Tunnel connects to the untrusted part of the system (Figure 2, §4), and meanwhile IP stacks have been hardened over decades. Furthermore, this lets us leverage existing software for remotely accessing USB devices over a network [60, 73, 106]. Our implementation uses usbredir [73], which (on the red machine), captures USB transfers, listens on a network socket, and uses a custom protocol to encapsulate USB transfers inside a TCP stream.

As a usbredir client, the Gateway receives usbredir packets, filters or modifies them, and then, playing the role of a usbredir server, delivers them to the QEMU process running the blue machine. A module in QEMU is the corresponding client; it decapsulates the USB transfers (using usbredir) and injects them into a virtual host controller created by QEMU and exposed to the blue machine. From the virtual host controller, the USB transfers travel into the blue machine’s HCI, with no software modifications on the blue machine.

Our implementation of Cinch supports USB versions through USB 3.

6.2 Gateway details

The Gateway is implemented in Rust [46]; it comprises about 8 kSLoC. Its major modules are parsers for usbredir packets and USB transfers, and a library that provides abstractions for creating new Policies. This library is inspired by the Click modular router [109] and provides domain-specific abstractions for USB (as examples, de-multiplexing usbredir packets into USB transfers and filtering those transfers). As in Click, the user organizes modules into chains where one module’s output is the next module’s input. Several such chains can be configured to operate in parallel. Users configure module chains with files in JSON format.

OS	exploit identifier	exploit description	prevention mechanism
Windows 8.1	01:01:00:C:4	Audio device with non-existent streaming interface	Signature Policy*
	01:01:00:C:5	Audio device with invalid streaming interface	Signature Policy*
	03:00:00:C:16	HID device with invalid report usage page	Compliance Policy
	03:00:00:C:17	HID device with invalid report usage page	Compliance Policy
	09:00:00:C:9	Hub with invalid number of ports	Compliance Policy
Linux 4.2.0	CVE-2016-2184	Sound device with non-existent endpoint	Assertion Policy
	CVE-2016-2185	RF remote control device with invalid interface or endpoint	Assertion Policy
	CVE-2016-2186	Multimedia control device with invalid endpoint	Assertion Policy
	CVE-2016-2187	Digitizer tablet device with invalid endpoint	Assertion Policy
	CVE-2016-2188	I/O Warrior device with invalid endpoint	Assertion Policy
	CVE-2016-2384	Audio device with invalid USB descriptor	Assertion Policy
	CVE-2016-2782	Serial device with no bulk-in or interrupt-in endpoint	Assertion Policy
	CVE-2016-3136	Serial device without two interrupt-in endpoints	Assertion Policy
	CVE-2016-3137	Serial device without both in and out interrupt endpoints	Assertion Policy
	CVE-2016-3138	Communication device without both control and data endpoints	Assertion Policy
	CVE-2016-3139	Drawing tablet with invalid USB descriptor	Assertion Policy
	CVE-2016-3140	Serial converter device with invalid USB descriptor	Assertion Policy
	CVE-2016-3951	Communication device with invalid descriptor and payload	Compliance Policy

*Exploit can be prevented with the compliance Policy, but we have not yet incorporated the necessary class specification (Audio) into Cinch.

FIGURE 3—Exploits for known-signature exercise (§7.1). Windows exploits were found by Boteanu and Fowler [79] with *umap* [88]; the reported identifier can be passed to *umap* using the “-s” flag to reproduce the exploit. We implemented the Linux exploits to target all USB-related CVEs from January–June 2016. The last column describes which Policy (§5) of Cinch prevents the exploit.

6.3 Proof-of-concept USB crypto adapter

We implement the crypto adapter (§5.4) using a Beagle-Bone Black [9] single-board computer that has a 1 GHz ARM Cortex-A8 processor and 512 MB RAM. For authentication, we generate a CA certificate and install it on the Gateway and crypto adapter. We use that CA certificate to sign certificates for the Gateway and crypto adapter, which mutually authenticate during the TLS handshake. The crypto adapter runs a version of *usbredir* that we augmented with support for TLS 1.2 [90] using OpenSSL [40]; these changes comprise less than 200 lines of code. The Gateway’s crypto module uses *stunnel* [49] to listen for TLS connections.

7 Evaluation

Our evaluation of Cinch answers the following questions:

- *How effectively does Cinch defend against attacks?* We subject Cinch to known exploits (§7.1), fuzzing (§7.2), and a red team exercise (§7.3).
- *Can new functionality be developed and deployed on Cinch with ease?* We answer this question qualitatively, by relating our experiences (§7.4).
- *What is Cinch’s performance overhead?* We examine latency and throughput (§7.5).

Experimental hardware and OSes. All of our experiments run on a single machine with a 3.3 GHz Intel i5-4590 and 16 GB of RAM. The hypervisor is Debian Jessie running Linux 4.2.0 with KVM enabled. The red machine’s OS is also Debian Jessie running Linux 4.2.0. The blue machine’s OS depends on the experiment and is either Windows 7 Ultimate SP1 (build 7601), Windows

8.1 Professional (build 9600), Debian Jessie with Linux 4.2.0, or Ubuntu 14.04 with a modified 4.2.0 kernel.

7.1 Known-signature attacks

We begin our evaluation of Cinch by subjecting it to synthetic attacks, based on documented vulnerabilities. For the attacks that succeed, we specify a “rematch” protocol, in which the operator can install a signature (§5.1) and then retry. This exercise is intended to address a counterfactual hypothetical: if Cinch had been deployed at the time of these vulnerabilities, would it have protected against their exploitation? And, if not, would a subsequent defensive reaction have been effective?

Method and experiment. We filter the CVE database [14] to select all the USB-related vulnerabilities reported from January to June of 2016. The resulting 13 CVEs apply to Linux 4.5 and earlier. For each CVE, we construct a payload that exploits it. We also include five exploits, disclosed by Boteanu and Fowler [79], that affect the most recent version of Windows 8.1; the targeted vulnerabilities are not in the CVE database.

Figure 3 summarizes the exploits. We confirm that each exploit successfully compromises the blue machine (Debian Jessie with Linux 4.2.0 or Windows 8.1) in the absence of Cinch. Once Cinch is enabled, we consider an attack successful if it compromises either the blue machine’s kernel or the Gateway.

On the offensive side, we mount the attacks using a Facedancer [98]—a custom USB microcontroller that can masquerade as any USB device and issue arbitrary payloads when connected to the target machine. We program

	exploits prevented	
	match phase	rematch phase
Known exploits (§7.1)		
Windows 8.1	3 / 5	5 / 5
Linux 4.2.0	13 / 13	13 / 13
vUSBf [136, 137] payloads (§7.2)		
randomized devices	10,000 / 10,000	N/A
sample exploits	13 / 13	N/A
red team round 1 (§7.3)		
Windows 7	2 / 2	2 / 2
Linux 4.2.0	3 / 5	5 / 5
red team round 2 (§7.3)		
Windows 7	3 / 3	3 / 3
Linux 4.2.0	11 / 16	13 / 16
red team round 3 (§7.3)		
Windows 7	3 / 3	3 / 3
Linux 4.2.0	15 / 20	16 / 20

FIGURE 4—Summary of Cinch’s security evaluation.

and control the Facedancer through a Python interface, using the GoodFET [99] and umap [88] tools.

On the defensive side, we configure Cinch with the signature, assertion, compliance, and logging Policies (§5). For the assertion Policy, we install 12 driver-specific configuration restrictions; these fix buggy or nonexistent checks, identified by the CVEs. For the signature Policy, we start with an empty signature database and check whether each attack succeeds; if it does, we craft a signature based on the payload and associated metadata, then conduct a rematch.

Results are summarized in Figure 4 (“Known exploits”); for each exploit, the mechanism that prevented it is listed in Figure 3. Cinch successfully detects and drops 16 offending payloads with no additional configuration. Two of the payloads were successful on their first try, but were blocked in the rematch phase; these payloads targeted vulnerabilities in the USB Audio class, which we have not yet included in Cinch’s compliance Policy.

7.2 Fuzzing

Next, we assess the robustness of Cinch’s compliance Policy (§5.2), via fuzz testing. We limit this exercise to attacks that target device enumeration, as implemented in the core and class drivers (§2). On the one hand, this is not a comprehensive exercise. At the same time, device enumeration is a common and well-studied source of vulnerabilities [137], accounting for about half of all USB-related entries in the CVE database.

In enumerating devices, USB core processes each device’s USB *descriptors*: records, generated by the device, that identify its manufacturer, function, USB version, capabilities, etc. This process is complex because of the

wide range of possible device configurations. Furthermore, the attack surface includes class driver initialization functions, since USB core passes descriptors to those functions; Schumilo et al. [137] demonstrate that many OSes and drivers do not handle device enumeration properly, especially when the device information is inconsistent or maliciously crafted.

Method and experiment. On the offensive side, we use vUSBf [136], a fuzzing tool that generates a random set of device descriptors and then emulates a device attach event. We update vUSBf to work with the most recent version of usbredir (v0.7.1), and we replace the red machine with an instance of vUSBf (that is, vUSBf communicates directly with the Gateway). In this setup, vUSBf can emulate hundreds of randomized devices per minute.

We run two experiments. In the first, we use vUSBf to emulate 10,000 randomly-generated devices. In the second, we use vUSBf to emulate 13 specific configurations identified by the vUSBf authors (after millions of trials) that crash some (older) systems.

On the defensive side, we run Cinch, configured with compliance (§5.2) and logging (§5.5) Policies. If Cinch allows the emulated device to communicate with the blue machine, we account this a failure.

We expect that the overwhelming majority of test cases will not obey the USB specification, and that Cinch’s compliance Policy will detect and prevent these cases. As a baseline, we also present the same 10,000 inputs to a system that is not running Cinch.

Results are summarized in Figure 4 (“vUSBf”). Cinch’s compliance module prevents all emulated devices from connecting to the blue machine. The three most commonly detected violations are: (1) improperly formatted strings, (2) invalid device classes, and (3) invalid or inconsistent number of functions. On the one hand, these results could be argued to be inconclusive because none of these inputs were successful against the baseline setup *without* Cinch. On the other hand, Cinch detected and blocked even the 13 configurations known to crash older systems.

7.3 Red team exercise

Our next set of exercises evaluates Cinch against attacks that were not known to us a priori. This is intended to assess Cinch’s effectiveness and to avoid some of the bias that may arise when developers choose the attack experiments (as above).

Specifically, we set up a red team that was charged with developing new USB exploits to compromise blue machines; this activity included crafting new vulnerabilities in the blue machine’s OS, which was meant to emulate the ongoing process of discovering and patching bugs. In our case, the red team comprised a subset of the authors who were kept separate from the developers of Cinch and

Protocol	There are three rounds, each of which has a <i>setup</i> , <i>match</i> and <i>rematch</i> phase. Setup: Red team chooses an OS (which they can modify arbitrarily) and develops exploits that crash the OS. Match: Cinch developers configure Cinch to run the OS provided by the red team as the blue machine; both teams confirm that the exploits crash the OS when Cinch is not present. The Cinch developers deploy Cinch, and the red team mounts its exploits. The Cinch developers collect traces, and both teams document the outcome of the exercise. Rematch: Cinch developers get the traces, and are given the opportunity to analyze and react to them. Then the match phase is rerun.
Attacker knowledge	Round 1: The red team is given access to a technical report that documents an earlier version of Cinch. This models an attacker with limited knowledge of Cinch. Round 2: The red team is given access to a machine that is running Cinch. This models an attacker with black-box access to Cinch, or an attacker that possesses Cinch’s binaries. Round 3: The red team is given access to Cinch’s source code. This models an attacker with full knowledge of Cinch’s logic (but not its configuration).
Developer ability	Cinch developers freeze Cinch’s code prior to the match phase of round 1. After that, Cinch developers may apply configuration-only changes: new signatures, etc.

FIGURE 5—Summary of the protocol for the red team exercise. This protocol was codified before the exercise began.

worked independently. Interactions between the red team and the developers were tightly controlled, following an evaluation protocol that was documented in advance. Figure 5 summarizes the protocol.

Summary of red team exploits. The red team developed 3 exploits for Windows and 20 exploits for Linux across the three rounds of the protocol. Some exploits shared the same attack vector but used different payloads.

The Windows exploits attacked a fresh copy of Windows 7; the red team did not install updates because the vulnerabilities their exploits targeted have been patched. Since red team members did not have access or visibility into the Windows USB stack, these exploits were found primarily through fuzzing, guided by past CVEs.

For Linux, the red team installed a modified version of kernel 4.2.0 on a fresh copy of Ubuntu 14.04. In particular, the red team modified a function within HCI that processes USB request blocks (the data structure representing a message in the USB subsystem) to trigger a kernel crash on certain device payloads; introduced a bug in USB core that causes the kernel to crash whenever a device with a certain configuration is connected; inserted a bug in Linux’s HID input subsystem (`drivers/input/input.c`) that leads to a null pointer dereference when it receives a specific sequence of input events; and introduced buggy drivers for a USB printer, camera, audio, and HID device.

Finally, the red team noticed that the VFAT filesystem driver in Linux 4.2 does not correctly validate the BIOS Parameter Block (BPB). While they were unable to exploit this bug directly, it can result in an invalid filesystem being mounted. To “enhance” this bug, the red team introduced a null pointer dereference in the BPB handling routine (`fs/fat/inode.c`), triggered by a filesystem with an invalid BPB.

Results are summarized in the last 3 sections of Figure 4.

First round. The red team developed 7 exploits for this round (2 for Windows and 5 for Linux). In the match phase, Cinch prevented both Windows exploits and 3 out of the 5 Linux exploits. The Windows exploits were prevented by Cinch’s architecture rather than by any of its Policies. Specifically, the red machine runs a Linux kernel; that kernel is not vulnerable to either of the Windows exploits and recognizes both connected devices as invalid. As a result, Cinch does not export these devices in the first place, protecting the Windows blue machine.

The two Linux exploits that Cinch was unable to prevent occurred at layers that were outside of its semantic knowledge (VFAT and the input subsystem). Using the traces—collected with Cinch’s logging module (§5.5)—the Cinch developers derived signatures. In the rematch phase, these signatures prevented the exploits.

Second round. In the match phase, Cinch prevented 14 out of 19 attacks, including attacks from the first round. The rematch phase again relied on signatures; of the remaining five exploits, signatures blocked two. The remaining three succeeded because they are polymorphic: they alter their payload to evade detection.

Third round. In the match phase, Cinch prevented 18 out of 23 attacks, including attacks from prior rounds for which signatures were available. In the rematch phase, Cinch was able to defend against an additional exploit using a signature that prevents a particular sequence of key presses from triggering a bug in the modified USB HID driver. The remaining four exploits are polymorphic and escaped evasion by signature and compliance checks.

These results, while preliminary, suggest that Cinch is able to prevent several exploits—primarily those that act as invalid USB devices—with prior configuration; several more can be prevented after deriving signatures. The remaining exploits might be prevented with more intrusive approaches (e.g., sandboxing; §5.3)

Tradeoff between security and availability. It is possible to develop more aggressive signatures to prevent polymorphic attacks (for example, using regular expressions); however, this risks disabling benign devices. To ensure that our signatures did not cause such false positives, we established a representative set of benign devices: a USB flash drive, printer, phone, SSD, keyboard, and mouse. After each phase of the experiment, we checked that our signatures did not keep these devices from working.

We found one failure: the signatures for the VFAT exploit prevented the blue machine from communicating with *any* storage device with a VFAT filesystem. We removed the offending signature and accounted that test a failure (i.e., Cinch did not prevent the exploit), since such a signature would not be deployable for most users.

7.4 Cinch’s flexibility and extensibility

There are two ways that Cinch can currently be extended: through new signatures and configurations to enhance existing Policies (§5), and through new Policies that add new functionality. We discuss our experience in both cases.

Deriving new signatures. We take a straightforward approach to deriving signatures for a given attack: we first log malicious traces, and then replay them in a controlled debugging environment. This allows us to analyze the configuration and the attack. We use this information to derive candidate signatures that are on the order of 10–15 lines of JSON; deriving a signature for the exploits in Section 7.3 took roughly 5 to 30 minutes, depending on: (1) the amount of data the exploit sent, and (2) the complexity of the subsystem the exploit targeted.

Creating new Policies. Adding a new Policy for Cinch requires implementing an instance of a Rust *trait* [2] (roughly analogous to a Java interface or a C++ abstract class; this trait is defined in the Gateway library, §6.2) that processes USB transfers, and adding the new Policy to Cinch’s configuration file. Based on this configuration, Cinch’s module subsystem automatically dispatches USB transfers to configured chains (§6.2). To give an idea of Policies’ complexity, Cinch’s largest—compliance—is 2500 SLoC while the rest average just 180 SLoC.

7.5 What are the costs of Cinch?

To understand the performance cost associated with using Cinch, we investigate two microbenchmarks, one for latency and one for throughput. We use Debian Jessie (Linux 4.2.0) as the blue machine’s OS.

Is Cinch’s added latency acceptable? To quantify the delay introduced by the components of Cinch, we connect the blue machine and another machine on a local network, using an Ethernet-over-USB adapter. We record the round-trip time between the two machines (using ping) as we

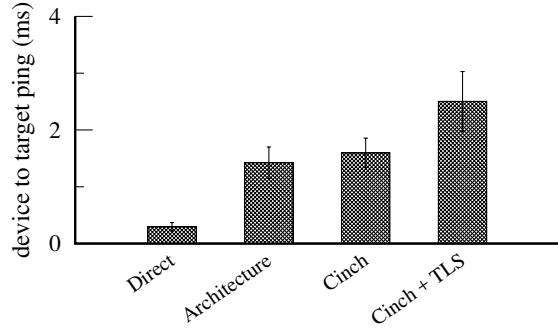


FIGURE 6—Round-trip time between the blue machine and USB device as components of Cinch are progressively added. Results are averaged over 1000 pings and error bars represent one standard deviation of the mean.

	direct	Cinch
USB 2 device (flash drive)		
% of CPU cycles	1.8 %	8.1%
memory	9 MB	205 MB
I/O throughput	181.6 Mbps	145.6 Mbps
Encrypted I/O throughput	–	35.4 Mbps
USB 3 device (SSD)		
% of CPU cycles	5.6%	38.2%
memory	9 MB	207 MB
I/O throughput	3.4 Gbps	2.1 Gbps

FIGURE 7—Resource consumption of Cinch when transferring a 1 GB file from storage devices to the blue machine. The “direct” baseline is a setup where devices are connected directly to the blue machine. Entries are the mean over 20 trials; standard deviation is less than 5%. We do not report encrypted throughput for the SSD because the crypto adapter does not support USB 3.

add components of Cinch. Figure 6 shows the results.

For our baseline, we connect the Ethernet-over-USB adapter directly to a USB port on the host (Fig. 6, “Direct”). We next attach the device to the red machine and export it to the blue machine through the Tunnel without the Gateway (i.e., the Tunnel runs directly to the blue machine); this arrangement demonstrates the latency cost of Cinch’s use of virtualization (Fig. 6, “Architecture”). Next, we add the Gateway to the above configuration, enabling all of Cinch’s Policies (§5), demonstrating the overhead when the Gateway interposes on all USB transfers (Fig. 6, “Cinch”). Finally, we place the crypto adapter (§5.4) in between the Ethernet-over-USB device and the Gateway (Fig. 6, “Cinch + TLS”).

Each component of Cinch adds moderate delay, with the full setup (including the crypto adapter) resulting in a round-trip time of less than 2.5 ms. We believe that this delay is acceptable for latency-sensitive input devices; as a comparison, high-performance mechanical keyboards introduce delays on the order of 5 ms between successive keystrokes (for debouncing [69, 107]).

What is Cinch’s impact on throughput and other resources? We read 1 GB of data from a USB storage device to the blue machine and measure the throughput, memory consumption, and CPU load with and without Cinch; we repeat these experiments 20 times. Storage devices range in performance, so we experiment with two: a USB 2 flash drive and a USB 3 SSD.

Figure 7 tabulates the results. For the flash drive, Cinch achieves $0.8\times$ the baseline’s throughput. There are two main reasons for this: (1) Cinch copies USB transfers at several stages in its architecture; and (2) USB 2 flash drives use exclusively synchronous transfers, meaning that Cinch’s added latency translates to lower throughput. For the USB 3 SSD, Cinch achieves $0.6\times$ the baseline’s throughput. Unlike in USB 2, USB 3 storage devices use asynchronous transfers and allow multiple in-flight requests. The primary overhead is thus memory copies.

With regard to CPU and memory use, Cinch has modest overhead. The memory Cinch consumes, which is primarily allocated to running the red machine, is in line with the cost of other security applications (e.g., antivirus).

7.6 Summary and critique

Our evaluation shows that Cinch can prevent previously documented vulnerabilities, fuzzing attempts, and crafted attacks, even without attack-specific configuration. Augmented with a signature database, its success is even higher, though none of its Policies are well suited to defeating polymorphic attacks. In this respect, Cinch is comparable to related tools in network security: it rules out certain classes of vulnerabilities and can be adapted to address specific issues, but it is not perfect. Cinch’s extensibility also seems reasonable, though our metrics here are subjective; and the performance impact, while not negligible, may be a good trade-off.

While this evaluation suggests that Cinch is a step in the right direction, it is far from definitive. First, we have likely not explored the full attack space, especially with regard to attacks on the non-USB portions of the kernel and on user software. Second, the red team comprised authors rather than disinterested parties, which may bias the security evaluation. Third, most systems are considered usable by their implementers; a neutral, non-expert operator may have a different perspective. Finally, Cinch’s performance impact may be acceptable for a wide range of devices, but others (e.g., audio and video devices) have more stringent latency requirements that Cinch might not meet, especially when using the crypto adapter.

8 Related work

Cinch’s contribution is architectural: most of its mechanisms are adapted from prior works and existing areas of research. Nevertheless, we are not aware of any other

system that addresses the full space of attacks described in Section 3.

USB security mechanisms (similar problem, different mechanisms). One can purchase an adapter that prevents data interchange on the USB bus, converting the bus into power lines only [51]. A software version of this protection is a set of Linux kernel patches known as grsecurity [23], which essentially disable hotplug. This “air gap ethos”—provide defense by eliminating connectivity—conflicts with Cinch’s aim of controlled interaction.

Qubes [45] is a distribution of Linux that makes extensive use of virtualization to create isolated privilege domains for *applications*. Qubes can place USB devices in their own virtual machines (USB VMs). A device’s transfers are delivered to its USB VM, and hence applications accessing that device need to live on that VM, wherein the threats enumerated in Section 3 are reprise. An exception is that Qubes allows a user to safely share USB storage devices from a USB VM with other VMs on the system by exporting them as block devices. Qubes also supports exporting keyboards and mice from a USB VM, but its developers warn that doing so risks exposing the system to attacks [62].

The udev user space daemon on Linux [56, 110] implements finer-grained policies than Qubes, akin to Cinch’s containment Policy (§5.3). However, udev can itself be attacked: udev requires the kernel to interact with every device that connects, so the device has an opportunity to attack the host machine before udev makes a policy decision. There are many commercial offerings that enable access control for USB devices [13, 15, 18, 22, 33, 35, 37, 48, 50, 55]; the issues with these are similar to udev.

USBFILTER [147] enables more precise and expressive access control policies than udev. Furthermore, these policies are enforced throughout the lifetime of the interaction rather than only at connection time. In particular, a user can define rules to dictate which entities (processes and drivers) can interact with a device (and vice versa). This is similar to Cinch’s containment Policy (§5.3), but USBFILTER’s rules support finer-grained statements, for example, restricting interaction to particular processes. The tradeoff is that it requires instrumenting the host’s OS to trace USB transfers all the way to the requesting processes and drivers. USBSec [149] brings a similar tradeoff: it extends the USB protocol with mutual authentication between the host and a compatible device (providing a subset of Cinch’s crypto Policy functionality; §5.4) but requires changes to the host’s USB stack.

GoodUSB [146] loads devices in a sandboxed environment and prompts the user to enable functions based on a device’s claimed identity. This is similar to (but richer than) Cinch’s containment Policy (§5.3), which could be enhanced accordingly. GoodUSB’s mechanisms might

also be used to bootstrap Cinch’s crypto overlay, as mentioned in Section 5.4.

Under UScramBle [124], devices provide a key to the host that can be used to encrypt further messages; the message goes upstream and thus is not broadcast across the bus (§3.3). This prevents eavesdropping for USB 2 and earlier, but unlike Cinch’s crypto overlay (§5.4), it cannot protect against malicious or compromised hubs that see the key.

Of the foregoing, only USBFILTER, USBSec, and GoodUSB address masquerading attacks (with the help of the user; §5.3); eavesdropping (§3.3) is out of scope for these systems. In contrast, UScramBle addresses eavesdropping but not masquerading.

Device driver isolation and reliability (complementary problem, overlapping mechanisms). There is a vast literature on device driver containment and reliability. We will go over some of it, but we can only scratch the surface (a helpful survey appears in SUD [80]). We note at the outset that Cinch borrows mechanisms from many of these works: placing drivers in a separate virtual machine [93, 95, 114], isolating a device with the IOMMU [105], and leveraging hardware-assisted I/O virtualization [105, 114, 145]. However, the threat and the resulting architecture are different.

Specifically, work that isolates faulty device drivers [80, 83, 93, 95, 96, 105, 112, 114, 127, 143–145, 152] assumes that hardware obeys its specification (and, with the exception of SUD [80], that drivers may be buggy, but not malicious). The same assumption about hardware is made by work that validates the commands passed to devices [152], eliminates bugs from drivers [130], and synthesizes drivers that are correct by construction [131, 132]. There is work that aims at tolerating hardware faults [108], but these faults are non-malicious and constrained (for example, flipped bits) compared to the types of attacks outlined in Section 3.

As a result of the assumption about faithful hardware, masquerading and eavesdropping are out of scope; often, devices that deviate from specification (§3.3) are, too. On the other hand, Cinch does not provide comprehensive protection against compromised drivers (though it can sanitize drivers’ inputs, as outlined in §5.2). For this reason, the works covered above are complementary to—and in many cases composable with—Cinch.

Secure peripheral interaction (different problem, overlapping mechanisms). Kells [82], USB Fingerprinting [76, 113], and work by Wang and Stavrou [150] allow a USB device to establish the identity of a host. The first two works are defense mechanisms *against* the host: they prevent compromised OSes from corrupting devices or propagating malware; the latter is an attack primitive and

allows a malicious device to compromise hosts selectively. Cinch’s crypto overlay (§5.4) also allows a device to identify a host (since connections can be mutually authenticated; §6.3), but the goal is to prevent eavesdropping and device masquerading.

SeRPEnT [151] and Bumpy [121] provide a safe pathway from devices, through an untrusted host machine, to a trusted, remote machine. SeRPEnT provides a similar abstraction to Cinch’s crypto overlay (§5.4), and its mechanism is comparable to Cinch’s crypto adapter. Bumpy’s goal, however, is remote attestation of user input rather than prevention of masquerading attacks; its mechanisms are based on trusted hardware. Both of these works target wide area networking, while Cinch focuses on intra-host communication.

Zhou et al. [155] allow trusted applications running on top of untrusted OSes to securely communicate with I/O devices. This is done via a trusted hypervisor that mediates access to hardware by both the trusted and untrusted components. Cinch also interacts with peripheral devices via an untrusted intermediary, but the architecture, mechanisms, goals, and threat model are all different.

Separation kernels and network security (related problems, related mechanisms). Two other research areas deserve special mention. The first is Rushby’s separation kernel [129], in which the operating system is architected to make a computer’s components interact as if they were part of a distributed system (see [81] and [122] for modern implementations). The foundational observation of this work—that networks are a useful abstraction for interposition—is one that we share. However, our goals and scenario are different. The separation kernel was intended to be a small kernel, with compartmentalized units that could be formally verified, and it provided separation through information flow control. In contrast, our scenario is commodity operating systems, and we are seeking to apply the conceptual framework of network security.

This brings us to network security itself. Cinch owes a substantial debt to this field, borrowing as it does concepts like firewalls, deep packet inspection, and virtual private networks. Moreover, the recent trend toward Network Function Virtualization (NFV) [119, 138] applies I/O virtualization (as do Cinch and some of the works cited earlier), but the point in NFV is to make middleboxes virtual, for reasons of configurability and cost.

9 Summary and conclusion

Cinch was motivated in large part by the observation that hardware security is recapitulating the history of network security. Originally, the Internet was a comparatively small number of mutually trusting organizations and users. As a consequence, there was relatively little focus on support for security within the network infras-

ture. With the explosion of Internet users, spurred by changing economics, security suddenly became a serious problem. Similarly, commodity operating systems have relatively few safeguards against misbehaving hardware, reflecting a time when peripheral devices could be trusted. But, with the rapid decline in the barriers to producing plug and play peripherals, those days have come to an end—and Cinch aims to be useful in the world ahead.

Although Cinch’s individual mechanisms have ample precedent in the literature, the architecture and the synthesis is novel, to the best of our knowledge. Moreover, as the evaluation results make clear, the implementation is pragmatic and surprisingly powerful. Looking at this fact, we feel comfortable stating that we have identified a good abstraction for the problem at hand.

To be clear, we are not saying that Cinch uniquely enables any one piece of its functionality (§5); rather, the abstraction makes it natural to develop and deploy what would require far more work under alternative solutions (§8).

We are also not saying that Cinch is comprehensive. Indeed, besides the limitations covered earlier (§1, §4.2, §7.6), some of Cinch’s solutions are effective only with additional mechanisms. As a key example, providing authentication and privacy with Cinch requires certificates or pairing, and device modifications. However, certificates are compatible with the chain of trust inherent in purchasing hardware, pairing is similar to the permissions model on mobile devices, and the required modifications are not onerous, as our implementation of the adapter (§6.3) indicates. As another example, Cinch’s compliance Policy (§5.2) would be strengthened by formal verification.

Despite the issues, Cinch appears to improve on the status quo. Of course, it is possible that, if Cinch were widely deployed, it would only escalate an arms race, and drive attackers to find ever more esoteric vulnerabilities. On the other hand, security is always about building higher fences, and the considerations at the heart of our work could guide the future design of peripheral buses and drivers.

Acknowledgements

This paper was aided by conversations with Andrew Baumann, Adam Belay, Sergio Benitez, Kevin Butler, Christian Huitema, Trammell Hudson, Ant Rowstron, Dennis Shasha, Jeremy Stribling, Ymir Vigfusson, and Junfeng Yang; and substantially improved by the detailed comments of the SOSP and USENIX Security reviewers. This work was supported by NSF grants CNS-1055057, CNS-1423249, and CNS-1514422; AFOSR grant FA9550-15-1-0302; and ONR grant N00014-14-1-0469.

References

- [1] 1394-2008—IEEE standard for a high-performance serial bus. <http://standards.ieee.org/findstds/standard/1394-2008.html>.
- [2] Abstraction without overhead: traits in Rust. <http://blog.rust-lang.org/2015/05/11/traits.html>.
- [3] Advanced Teensy penetration testing payloads. <https://www.offensive-security.com/offsec/advanced-teensy-penetration-testing-payloads/>.
- [4] AnywhereUSB5 integer overflow. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4459>.
- [5] Apple macbook tech specs. <http://www.apple.com/macbook/specs/>.
- [6] BadUSB—now with do-it-yourself instructions. <https://nakedsecurity.sophos.com/2014/10/06/badusb-now-with-do-it-yourself-instructions/>.
- [7] BadUSB: Big, bad USB security problems ahead. <http://www.zdnet.com/article/badusb-big-bad-usb-security-problems-ahead/>.
- [8] BadUSB: what you can do about undetectable malware on a flash drive. <http://www.pcworld.com/article/2840905/badusb-what-you-can-do-about-undetectable-malware-on-a-flash-drive.html>.
- [9] BeagleBone Black. <http://beagleboard.org/BLACK>.
- [10] Chromebook pixel. <http://www.google.com/chromebook/pixel/>.
- [11] ClamAV. <http://www.clamav.net/>.
- [12] Close access SIGADS. <https://www.documentcloud.org/documents/807030-ambassade.html#document/p1>.
- [13] CoCoSys Endpoint Protector. http://www.endpointprotector.com/products/endpoint_protector.
- [14] Common vulnerabilities and exposures. <https://cve.mitre.org>.
- [15] Comodo Endpoint Security Manager. <https://www.comodo.com/business-enterprise/endpoint-protection/endpoint-security-manager.php>.
- [16] COTTONMOUTH-I. <https://nsa.gov1.info/dni/nsa-ant-catalog/usb/index.html#COTTONMOUTH-I>.
- [17] COTTONMOUTH-II. <https://nsa.gov1.info/dni/nsa-ant-catalog/usb/index.html#COTTONMOUTH-II>.
- [18] DeviceLock Data Loss Prevention Suite. <http://www.devicelock.com/products/>.
- [19] DLL planting remote code execution vulnerability. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0096>.
- [20] Equation group: Questions and answers. https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf.
- [21] Equation: The Death Star of Malware Galaxy. <https://securelist.com/blog/research/68750/equation-the-death-star-of-malware-galaxy/>.
- [22] GFI EndpointSecurity. <http://www.gfi.com/products-and-solutions/network-security-solutions/gfi-endpointsecurity>.
- [23] grsecurity. <https://grsecurity.net>.
- [24] Hackers pierce network with jerry-rigged mouse. http://www.theregister.co.uk/2011/06/27/mission_impossible_mouse_attack.
- [25] Hubs—BadUSB exposure. <https://opensource.srlabs.de/projects/badusb/wiki/Hubs>.
- [26] Inception. <https://github.com/carmaa/inception>.
- [27] Inside TAO: Documents reveal top NSA hacking unit. <http://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on>

- global-networks-a-940969.html.
- [28] Linux audio driver dereferences null pointer under invalid device. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2184>.
- [29] Linux default configuration does not warn user before enabling HID over USB. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0640>.
- [30] Linux serial driver dereferences null pointer under device with no bulk-in or interrupt-in endpoints. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2782>.
- [31] Linux hid-piclcd_core.c buffer overflow. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3186>.
- [32] Linux report_fixup HID functions out-of-bounds write. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3184>.
- [33] Lumension Device Control. <https://www.lumension.com/device-control-software/usb-security-protection.aspx>.
- [34] MatrixSSL open source embedded SSL and TLS. <http://www.matrixssl.org>.
- [35] McAfee Complete Data Protection. <http://www.mcafee.com/us/products/complete-data-protection.aspx>.
- [36] NanoSSL—an SSL library for embedded devices. <http://www.mocana.com/iot-security/nanossal>.
- [37] Novell ZENworks Endpoint Security Management. <https://www.novell.com/products/zenworks/endpointsecuritymanagement/>.
- [38] NSA reportedly installing spyware on US-made hardware. <http://www.cnet.com/news/nsa-reportedly-installing-spyware-on-us-made-hardware/>.
- [39] Only half of USB devices have an unpatchable flaw, but no one knows which half. <http://www.wired.com/2014/11/badusb-only-affects-half-of-usb/>.
- [40] OpenSSL. <https://www.openssl.org>.
- [41] OS X does not warn user before enabling HID over USB. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0639>.
- [42] OS X USB hub descriptor memory corruption. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3723>.
- [43] Phision 2251-03 (2303) custom firmware & existing firmware patches (BadUSB). <https://github.com/adamcaudill/Psychson>.
- [44] QEMU usb_host_handle_control function buffer overflow. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0297>.
- [45] Qubes OS project. <https://www.qubes-os.org>.
- [46] The Rust programming language. <https://www.rust-lang.org/>.
- [47] Snort.Org. <https://www.snort.org/>.
- [48] Sophos Endpoint Security and Control. <http://www.sophos.com/en-us/support/documentation/endpoint-security-and-control-for-windows.aspx>.
- [49] Stunnel. <http://www.stunnel.org>.
- [50] Symantec Endpoint Protection. <http://www.symantec.com/endpoint-protection/>.
- [51] SyncStop. <http://syncstop.com>.
- [52] Teensy USB development board. <https://www.pjrc.com/teensy>.
- [53] This thumbdrive hacks computers. <http://arstechnica.com/security/2014/07/this-thumbdrive-hacks-computers-badusb-exploit-makes-devices-turn-evil/>.
- [54] Thunderbolt technology. <http://www.intel.com/content/dam/doc/technology-brief/thunderbolt-technology-brief.pdf>.
- [55] Trend Micro Enterprise Data Protection. <http://www.trendmicro.com/us/enterprise/data-protection/endpoint/>.
- [56] udev. <http://www.freedesktop.org/software/systemd/man/udev.html>.
- [57] Universal Serial Bus revision 2.0 specification. http://www.usb.org/developers/docs/usb20_docs/usb_20_031815.zip.
- [58] Universal Serial Bus revision 3.1 specification. http://www.usb.org/developers/docs/usb_31_031815.zip.
- [59] USB device class specifications. http://www.usb.org/developers/docs/devclass_docs/.
- [60] USB over network. <http://www.usb-over-network.com>.
- [61] USB Rubber Ducky. <http://usbrubberducky.com>.
- [62] Using and Managing USB devices. Qubes OS Project. <https://www.qubes-os.org/doc/usb/>.
- [63] Windows crafted .LNK or .PIF arbitrary code execution. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2568>.
- [64] Windows disk partition driver elevation of privilege vulnerability. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-4115>.
- [65] Windows does not warn user before enabling HID over USB. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0638>.
- [66] wolfSSL. <http://www.yassl.com>.
- [67] Bluetooth user interface flow diagrams for Bluetooth secure simple pairing devices. Technical report, Bluetooth Usability Expert Group, Sept. 2007.
- [68] AMD-V nested paging. Technical report, AMD, July 2008.
- [69] Cherry MX series keyswitch, 2014. <http://cherrycorp.com/product/mx-series/>.
- [70] Intel virtualization technology for directed I/O, Oct. 2014. <http://www.intel.com/content/www/us/en/embedded/technology/virtualization/vt-directed-io-spec.html>.
- [71] AMD I/O virtualization technology (IOMMU) specification, Feb. 2015. http://support.amd.com/TechDocs/48882_IOMMU.pdf.
- [72] TURNIPSCHOOL - an open source reimagining of COTTONMOUTH-I, 2015. <https://github.com/mossmann/cc11xx/tree/master/turnipschool>.
- [73] usbredir, 2015. <https://github.com/SPICE/usbredir>.
- [74] C. Arthur. China's Huawei and ZTE pose national security threat, says US committee. <http://www.theguardian.com/technology/2012/oct/08/china-huawei-zte-security-threat>.
- [75] D. Barrall and D. Dewey. "Plug and Root," the USB key to the kingdom. In *Proceedings of the Black Hat USA Conference*, July 2005.
- [76] A. Bates, R. Leonard, H. Pruse, D. Lowd, and K. R. B. Butler. Leveraging USB to establish host identity using commodity devices. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2014.
- [77] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe, user-level access to privileged CPU features. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.
- [78] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.
- [79] D. Boteanu and K. Fowler. Bypassing self-encrypting drives

- (SED) in enterprise environments. In *Proceedings of the Black Hat Europe Conference*, Nov. 2015.
- [80] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, June 2010.
- [81] R. Buerki and A.-K. Rueeggseger. Muen—an x86/64 separation kernel for high assurance. Technical report, University of Applied Sciences Rapperswil (HSR), Switzerland, Aug. 2013. <http://muen.code-labs.ch/muen-report.pdf>.
- [82] K. R. B. Butler, S. E. McLaughlin, and P. D. McDaniel. Kells: A protection framework for portable data. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2010.
- [83] S. Butt, V. Ganapathy, M. M. Swift, and C.-C. Chang. Protecting commodity operating system kernels from vulnerable device drivers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2009.
- [84] A. Caudill. Making BadUSB work for you. <https://adamcaudill.com/2014/10/02/making-badusb-work-for-you-derbycon/>.
- [85] A. Crenshaw. Plug and Prey: Malicious USB devices. In *Proceedings of ShmooCon*, Jan. 2011.
- [86] Dark Purple. USB killer. <http://kukuruku.co/hub/diy/usb-killer>, 2015.
- [87] A. Davis. Lessons learned from 50 bugs: Common USB driver vulnerabilities. Technical report, NCC Group, Jan. 2013.
- [88] A. Davis. umap: the USB host security assessment tool, 2014. <https://github.com/nccgroup/umap>.
- [89] A. Davis. USB attacks need physical access right? Not any more.... In *Proceedings of the Black Hat Asia Conference*, Mar. 2014.
- [90] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.
- [91] C. Doctorow. Dropped infected USB in the company parking lot as a way of getting malware onto the company network. <http://boingoing.net/2012/07/10/dropped-infected-usb-in-the-co.html>.
- [92] R. Dominguez Vega. USB attacks: Fun with Plug and Own. In *Proceedings of the DEF CON Hacking Conference*, Aug. 2009.
- [93] Ú. Erlingsson, T. Roeder, and T. Wobber. Virtual environments for unreliable extensions. Technical Report MSR-TR-05-82, Microsoft Research, June 2005.
- [94] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet dossier. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.
- [95] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Oct. 2004.
- [96] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2008.
- [97] D. Goodin. Photos of an NSA “upgrade” factory show Cisco router getting implant. <http://arstechnica.com/tech-policy/2014/05/photos-of-an-nsa-upgrade-factory-show-cisco-router-getting-implant/>.
- [98] T. Goodspeed. Facedancer21. <http://goodfet.sourceforge.net/hardware/facedancer21/>.
- [99] T. Goodspeed. GoodFET. <https://github.com/travisgoodspeed/goodfet>.
- [100] T. Goodspeed. Emulating USB devices with Python, July 2012. <http://travisgoodspeed.blogspot.com/2012/07/emulating-usb-devices-with-python.html>.
- [101] G. Greenwald. How the NSA tampers with US-made internet routers. <http://www.theguardian.com/books/2014/may/12/glenn-greenwald-nsa-tampers-us-internet-routers-snowden>.
- [102] J. A. Halderman and E. W. Felten. Lessons from the Sony CD DRM episode. In *Proceedings of the USENIX Security Symposium*, Aug. 2006.
- [103] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the USENIX Security Symposium*, Aug. 2001.
- [104] F. Hao, M. Kodialam, T. V. Lakshman, and K. P. N. Puttaswamy. Protecting cloud data using dynamic inline fingerprint checks. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2013.
- [105] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2009.
- [106] T. Hirofuchi, E. Kawai, K. Fujikawa, and H. Sunahara. USB/IP—a peripheral bus extension for device sharing over IP network. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, Apr. 2005.
- [107] P. Horowitz and W. Hill. *The Art of Electronics*, chapter 9, Digital Meets Analog: Switch Bounce, pages 576–577. Cambridge University Press, 2nd edition, 1989.
- [108] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.
- [109] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3), Aug. 2000.
- [110] G. Kroah-Hartman. udev – a userspace implementation of devfs. In *Proceedings of the Ottawa Linux Symposium*, July 2003.
- [111] B. Lampson. Accountability and freedom. <http://research.microsoft.com/en-us/um/people/blampson/slides/accountabilityandfreedomabstract.htm>, 2005.
- [112] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20, 2005.
- [113] L. Letaw, J. Pletcher, and K. Butler. Host identification via usb fingerprinting. In *Proceedings of the IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)*, May 2011.
- [114] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.
- [115] Y. Liu, C. Corbett, K. Chiang, R. Archibald, B. Mukherjee, and D. Ghosal. SIDD: A framework for detecting sensitive data exfiltration by an insider attack. In *Proceedings of the Hawaii International Conference on System Sciences*, Jan. 2009.
- [116] F. Lone Sang, V. Nicomette, and Y. Deswarté. I/O attacks in Intel PC-based architectures and countermeasures. In *Proceedings of the SysSec Workshop*, July 2011.
- [117] K. S. Long. Catching the cyber spy: ARL’s interrogator. Technical Report ADA432198, Army Research Laboratory, Dec. 2004.
- [118] A. Mamiit. How bad is BadUSB? security experts say there is no quick fix. <http://www.techtimes.com/articles/17078/20141004/how-bad-is-badusb-security-experts-say-there-is-no-quick-fix.htm>.
- [119] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*

- (NSDI), Apr. 2014.
- [120] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham. Mouse trap: Exploiting firmware updates in USB peripherals. In *Proceedings of the USENIX Workshop on Offensive Technologies*, Aug. 2014.
- [121] J. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2009.
- [122] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2013.
- [123] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.
- [124] M. Neugschwandtner, A. Beitler, and A. Kurmus. A transparent defense against USB eavesdropping attacks. In *Proceedings of the European Workshop on System Security (EUROSEC)*, Apr. 2016.
- [125] K. Nohl and J. Lell. BadUSB—on accessories that turn evil. In *Proceedings of the Black Hat USA Conference*, Aug. 2014.
- [126] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.
- [127] M. J. Renzelmann and M. M. Swift. Decaf: Moving device drivers to a modern language. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, June 2009.
- [128] M. Rushanan and S. Checkoway. Run-DMA. In *Proceedings of the USENIX Workshop on Offensive Technologies*, Aug. 2015.
- [129] J. Rushby. The design and verification of secure systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1981.
- [130] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Mar. 2009.
- [131] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.
- [132] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.
- [133] K. Scarfone and P. Mell. Guide to intrusion detection and prevention systems (IDPS). Technical report, NIST, Feb. 2007.
- [134] N. Schear, C. Kintanna, Q. Zhang, and A. Vahdat. Glavlit: Preventing exfiltration at wire speed. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2006.
- [135] B. Schneier. Yet another “people plug in strange USB sticks” story. https://www.schneier.com/blog/archives/2011/06/yet_another_peo.html.
- [136] S. Schumilo. virtual USB fuzzer, 2015. <https://github.com/schumilo/vUSBf/>.
- [137] S. Schumilo, R. Spenneberg, and H. Schwartke. Don’t trust your USB! How to find bugs in USB device drivers. In *Proceedings of the Black Hat Europe Conference*, Oct. 2014.
- [138] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2012.
- [139] R. Sevinsky. Funderbolt: Adventures in Thunderbolt DMA attacks. In *Proceedings of the Black Hat USA Conference*, July 2013.
- [140] G. Silowash and T. Lewellen. Insider threat control: Using universal serial bus (USB) device auditing to detect possible data exfiltration by malicious insiders, 2013. CMU/SEI-2013-TN-003, <http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=35427>.
- [141] S. Stecklow. U.S. nuclear lab removes Chinese tech over security fears. <http://www.reuters.com/article/2013/01/07/us-huawei-alamos-idUSBRE90608B20130107>.
- [142] P. Stewin and I. Bystrov. Understanding DMA malware. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2012.
- [143] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4), 2006.
- [144] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1), 2005.
- [145] L. Tan, E. M. Chan, R. Farivar, N. Mallick, J. C. Carlyle, F. M. David, and R. H. Campbell. iKernel: Isolating buggy and malicious device drivers using hardware virtualization support. In *Proceedings of the IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC)*, Sept. 2007.
- [146] D. Tian, A. Bates, and K. Butler. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2015.
- [147] J. Tian, N. Scaife, A. Bates, K. R. B. Butler, and P. Traynor. Making USB great again with USBFILTER. In *Proceedings of the USENIX Security Symposium*, Aug. 2016.
- [148] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey. Users really do plug in USB drives they find. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2016.
- [149] Z. Wang, R. Johnson, and A. Stavrou. Attestation & authentication for USB communications. In *Proceedings of the IEEE International Conference on Software Security and Reliability Companion*, June 2012.
- [150] Z. Wang and A. Stavrou. Exploiting smart-phone USB connectivity for fun and profit. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2010.
- [151] D. Weinstein, X. Kovah, and S. Dyer. SeRPEndT: Secure remote peripheral encryption tunnel. Technical Report MP120013, The MITRE Corporation, 2012.
- [152] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2008.
- [153] R. Wojtczuk. Subverting the Xen hypervisor. In *Proceedings of the Black Hat USA Conference*, Aug. 2008.
- [154] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. On the DMA mapping problem in direct device assignment. In *Proceedings of the ACM International Systems and Storage Conference (SYSTOR)*, May 2010.
- [155] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2014.

Making USB Great Again with USBFILTER

Dave (Jing) Tian*, Nolen Scaife*, Adam Bates†, Kevin R. B. Butler*, and Patrick Traynor*

* University of Florida, Gainesville, FL

† University of Illinois, Urbana-Champaign, IL

{daveti, scaife, adammbates, butler, traynor}@ufl.edu

Abstract

USB provides ubiquitous plug-and-play connectivity for a wide range of devices. However, the complex nature of USB obscures the true functionality of devices from the user, and operating systems blindly trust any physically-attached device. This has led to a number of attacks, ranging from hidden keyboards to network adapters, that rely on the user being unable to identify all of the functions attached to the host. In this paper, we present USBFILTER, which provides the first packet-level access control for USB and can prevent unauthorized interfaces from successfully connecting to the host operating system. USBFILTER can trace individual USB packets back to their respective processes and block unauthorized access to any device. By instrumenting the host’s USB stack between the device drivers and the USB controller, our system is able to filter packets at a granularity that previous works cannot — at the lowest possible level in the operating system. USBFILTER is not only able to block or permit specific device interfaces; it can also restrict interfaces to a particular application (e.g., only Skype can access my webcam). Furthermore, our experimental analysis shows that USBFILTER introduces a negligible (3-10 μ s) increase in latency while providing mediation of all USB packets on the host. Our system provides a level of granularity and extensibility that reduces the uncertainty of USB connectivity and ensures unauthorized devices are unable to communicate with the host.

1 Introduction

The Universal Serial Bus (USB) provides an easy-to-use, hot-pluggable architecture for attaching external devices ranging from cameras to network interfaces to a single host computer. USB ports are pervasive; they can often be found on the front, back, and inside of a common desktop PC. Furthermore, a single USB connector may connect multiple device classes. These composite

devices allow disparate hardware functions such as a microphone and speakers to appear on the same physical connector (e.g., as provided by a headset). In the host operating system, technologies such as USBIP [21] provide the capability to remotely connect USB devices to a host over a network. The result is a complex combination of devices and functionalities that clouds the user’s ability to reason about what is actually connected to the host.

Attacks that exploit this uncertainty have become more prevalent. Firmware attacks such as BadUSB [27] modify benign devices to have malicious behavior (e.g., adding keyboard emulation to a storage device or perform automatic tethering to another network). Hardware attacks [1] may inject malware into a host, provide RF remote control capabilities, or include embedded proxy hardware to inject and modify USB packets. Attackers may also exfiltrate data from the host by leveraging raw I/O (e.g., using libusb [14]) to communicate with the USB device directly, or bypass the security mechanism employed by the USB device controller by sending specific USB packets to the device from the host USB controller [4]. Unfortunately, the USB Implementers Forum considers defending against malicious devices to be the responsibility of the user [44], who is unlikely to be able to independently verify the functionality and intent of every device simply by its external appearance, and may just plug in USB devices to take a look [43].

Modern operating systems abstract USB authorization to physical control, automatically authorizing devices connected to the host, installing and activating drivers, and enabling functionality. We believe that a finer-grained control over USB is required to protect users. In this paper, we make the following contributions:

- **Design and develop a fine-grained USB access control system:** We introduce USBFILTER, a packet-level firewall for USB. Our system is the first to trace individual USB packets back to the source or destination process and interface. USBFILTER

rules can stop attacks on hosts by identifying and dropping unwanted USB packets before they reach their destination in the host operating system.

- **Implement and characterize performance:** We demonstrate how USBFILTER imposes minimal overhead on USB traffic. As a result, our system is well-suited for protecting any USB workload.
- **Demonstrate effectiveness in real-world scenarios:** We explore how USBFILTER can be used to thwart attacks and provide security guarantees for benign devices. USBFILTER can pin devices (e.g., webcams) to approved programs (e.g., Skype, Hangouts) to prevent malicious software on a host from enabling or accessing protected devices.

USBFILTER is different from previous works in this space because it enables the creation of rules that explicitly allow or deny functionality based on a wide range of features. GoodUSB [41] relies on the user to explicitly allow or deny specific functionality based on what the device reports, but cannot enforce that the behavior of a device matches what it reports. SELinux [35] policies and PinUP [13] provide mechanisms for pinning processes to filesystem objects, but USBFILTER expands this by allowing individual USB packets to be associated with processes. This not only allows our system to permit pinning devices to processes, but also individual interfaces of composite devices.

Our policies can be applied to differentiate individual devices by identifiers presented during device enumeration. These identifiers, such as serial number, provide a stronger measure of identification than simple product and vendor codes. While not a strong authentication mechanism, USBFILTER is able to perform filtering without additional hardware. The granularity and extensibility of USBFILTER allows it to perform the functions of existing filters [41] while permitting much stronger control over USB devices.

The remainder of this paper is structured as follows: In Section 2, we provide background on the USB protocol and explain why it is not great anymore; in Section 3, we discuss the security goals, design and implementation of our system; in Section 4, we discuss how USBFILTER meets our required security guarantees; in Section 5, we evaluate USBFILTER and discuss individual use cases; in Section 6, we provide additional discussion; in Section 7, we explore related work; and in Section 8, we conclude.

2 Background

A USB device refers to a USB transceiver, USB hub, host controller, or peripheral device such as a human-interface

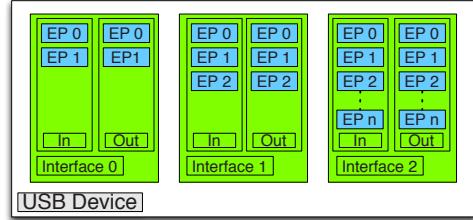


Figure 1: A detailed view of a generic USB device. Similar to a typical USB headset, this device has three interfaces and multiple endpoints.

device (*HID*, e.g., keyboard and mouse), printer, or storage. However, the device may have multiple functions internally, known as *interfaces*. An example device with three interfaces is shown in Figure 1. USB devices with more than one interface are known as composite devices. For example, USB headsets often have at least three interfaces: the speaker, the microphone, and the volume control functionalities. Each interface is treated as an independent entity by the host controller. The operating system loads a separate device driver for each interface on the device.

The USB protocol works in a master-slave fashion, where the host USB controller is responsible to poll the device both for requests and responses. When a USB device is attached to a host machine, the host USB controller queries the device to obtain the *configurations* of the device, and activates a single configuration supported by the device. For instance, when a smartphone is connected with a host machine via USB, users can choose it to be a storage or networking device. By parsing the current active configuration, the host operating system identifies all the *interfaces* contained in the configuration, and loads the corresponding device drivers for each interface. This whole procedure is called *USB enumeration* [10]. Once a USB device driver starts, it first parses the *endpoints* information embedded within this interface as shown in Figure 1.

While the interface provides the basic information for the host operating system to load the driver, the *endpoint* is the communication unit when a driver talks with the USB device hardware. Per specification, the endpoint 0 (EP0) should be supported by default, enabling *Control* (packet) transfer from a host to a device to further probe the device, prepare for data transmission, and check for errors. All other endpoints can be optional though there is usually at least EP1, providing *Isochronous*, *Interrupt*, or *Bulk* (packet) transfers, which are used by audio/video, keyboard/mouse, and storage/networking devices respectively. All endpoints are grouped into either *In* pipes, where transfers are from the device to the host,

or *Out* pipes, where transfers are from the host to the device. This in/out pipe determines the transmission direction of a USB packet. With all endpoints set up, the driver is able to communicate with the device hardware by submitting USB packets with different target endpoints, packet types, and directions. These packets are delivered to the host controller, which calls the controller hardware to encode USB packets into electrical signals and send them to the device.

2.1 Why USB Was Great

Prior to USB’s introduction in the 1990s, personal computers used a number of different and often platform-specific connectors for peripherals. Serial and parallel ports, PS/2, SCSI, ADB, and others were often not hot-pluggable and required users to manually set configuration options (such as the SCSI ID). The widespread industry adoption of USB fixed many of these issues by providing a common specification for peripherals. Hardware configuration is now handled exclusively by the host, which is able to manage many devices on a single port. The relative ease with which a USB peripheral can be installed on a host is simultaneously its greatest and most insecure property.

The USB subsystem has been expanded in software as well, with Virtio [30] supporting I/O virtualization in KVM, enabling virtual USB devices in VMs, and passing through the physical devices into VMs. USBIP [21] transfers USB packets via IP, making remote USB device sharing possible. Wireless USB (WUSB) [19] and Media Agnostic USB (MAUSB) [16] promote the availability of USB devices by leveraging different wireless communication protocols, making the distinction among local USB devices, virtual ones, and remote ones vanish.

Overall, the utility and complexity of USB has been steadily increasing in both hardware and software. Advances in circuit and chip design now allow hidden functionality to be placed inside the USB plug [1]. The ease-of-use that made USB great now threatens users by obscuring the individual interfaces in a USB device.

2.2 How USB Lost its Greatness

Attacks on USB prey on the fundamental misunderstanding of how devices are constructed from interfaces. Attacks such as BadUSB [27] and TURNIPSCHOOL [1] (itself designed on specifications from nation-state actors) use composite devices to present multiple interfaces to a host. Often these include one benign or expected interface and one or more malicious interfaces, including keyboards [9, 27] and network interfaces [27, 1]. Without communicating with the host operating system, a malicious USB device can only obtain power from the

host. While it may be possible to perform power analysis attacks without sending USB packets, we focus on the problem of connecting malicious devices to the host’s operating system. All of these attacks share a common thread: they attach an unknown interface to a host without the user’s knowledge. Since operating systems implicitly trust any device attached, these hidden functions are enumerated, their drivers are loaded, and they are granted access to the host with no further impediment.

Data exfiltration from host machines may be the main reason why USB storage is banned or restricted in enterprise and government environments. Current secure storage solutions rely on access control provided by the host operating system [23] or use network-based device authentication [22]. While access controls can be bypassed by raw I/O, which communicates to the device directly from userspace (e.g., using libusb [14]), network-based methods are vulnerable to network spoofing (e.g., ARP spoofing [32] and DNS spoofing [36]). It is thus unclear whether data exfiltration has occurred or not until the USB port is glued or locked [39]. The remainder of this paper will show how a packet-level filter for USB permits fine-grained access controls, eliminating the implicit trust model while providing strong guarantees.

3 USB Access Control

The complex nature of the USB protocol and the variety of devices that can be attached to it makes developing a robust and efficient access control mechanism challenging. Layers in the operating system between the process and the hardware device create difficulties when identifying processes. Accordingly, developing a system such as USBFILTER is not as simple as intercepting USB packets and dropping those that match rules. In this section, we discuss our security goals, design considerations, and implementation of USBFILTER while explaining the challenges of developing such a system.

3.1 Threat and Trust Models

We consider an adversary against our system who has restricted external physical or full network access to a given host. The adversary may launch physical attacks such as attaching unauthorized USB devices to the host system or tampering with the hardware of previously-authorized devices to add additional functionality. The physically-present adversary may not open the device or tamper with the internal storage, firmware, or any other hardware. This type of adversary might (for example) be present in an data center or retail location, where devices have exposed USB ports, but tampering with the chassis of the device would raise suspicion or sound alarms. The adversary may also launch network attacks in order

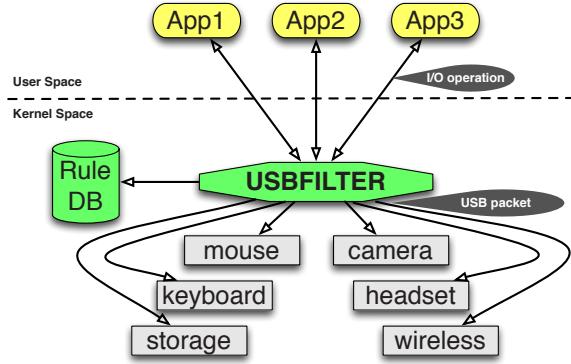


Figure 2: USBFILTER implements a USB-layer reference monitor within the kernel, by filtering USB packets to different USB devices to control the communications between applications and devices based on rules configured.

to enable or access authorized devices from unauthorized processes or devices. In either case, the adversary may attempt to exfiltrate data from the host system via both physical and virtual USB devices.

We consider the following actions by an adversary:

- **Device Tampering:** The adversary may attempt to attach or tamper with a previously-authorized device to add unauthorized functionality (e.g., BadUSB [27]).
- **Unauthorized Devices:** Unauthorized devices attached to the system either physically or virtually [21] can be used to discreetly interact with the host system or to provide data storage for future exfiltration.
- **Unauthorized Access:** The adversary may attempt to enable or access authorized devices on a host (e.g., webcam, microphone, etc.) via unauthorized software to gain access to information or functionality that would otherwise be inaccessible.

We assume that as a kernel component, the integrity of USBFILTER depends on the integrity of the operating system and the host hardware (except USB devices). Code running in the kernel space has unrestricted access to the kernel’s memory, including our code, and we assume that the code running in the kernel will not tamper with USBFILTER. We discuss how we ensure runtime and platform integrity in our experimental setup in Section 3.4.

3.2 Design Goals

Inspired by the Netfilter [40] framework in the Linux kernel, we designed USBFILTER to enable administrator-

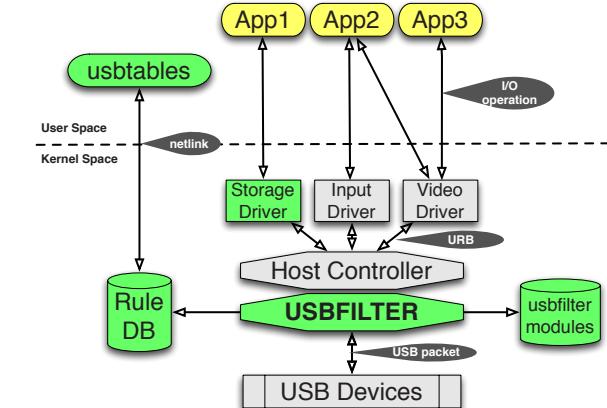


Figure 3: The architecture of USBFILTER.

defined rule-based filtering for the USB protocol. To achieve this, we first designed our system to satisfy the concept of a reference monitor [2], shown in Figure 2. While these goals are not required for full functionality of USBFILTER, we chose to design for stronger security guarantees to ensure that processes attempting to access hardware USB devices directly would be unable to circumvent our system. We define the specific goals as follows:

G1 Complete Mediation. All physical or virtual USB packets must pass through USBFILTER before delivery to the intended destination.

G2 Tamperproof. USBFILTER may not be bypassed or disabled as long as the integrity of the operating system is maintained.

G3 Verifiable. The user-defined rules input into the system must be verifiably correct. These rules may not conflict with each other.

While the above goals support the security guarantees that we want USBFILTER to provide, we expand upon these to provide additional functionality:

G4 Granular. Any mutable data in a USB packet header must be accessible by a user-defined rule. If the ultimate destination of a packet is a userspace process, USBFILTER must permit the user to specify the process in a rule.

G5 Modular. USBFILTER must be extensible and allow users to provide submodules to support additional types of analysis.

3.3 Design and Implementation

The core USBFILTER component is statically compiled and linked into the Linux kernel image, which hooks the

flow of USB packets before they reach the USB host controller which serves the USB device drivers, as shown in Figure 3. Like Netfilter, this USB firewall checks a user-defined rule database for each USB packet that passes through it and takes the action defined in the first matching rule. A user-space program, USBTABLES, provides mediated read/write access to the rule database. Since USBFILTER intercepts USB packets in the kernel, it can control access to both physical and virtual devices.

3.3.1 Packet Filtering Rules

To access external USB devices, user-space applications request I/O operations which are transformed into USB request blocks (URBs) by the operating system. The communication path involves the process, the device, and the I/O request itself (*USB packet*). Similarly, a USBFILTER rule can be described using the process information, the device information, and the USB packet information.

A USBFILTER rule R can be expressed as a triple (N, \mathcal{C}, A) where N is the name of the rule, \mathcal{C} is a set of conditions, and $A \in \{\text{ALLOW}, \text{DROP}\}$ is the action that is taken when all of the conditions are satisfied. As long as the values in conditions, action, and name are valid, this rule is valid, but may not be correct considering other existing rules. We discuss verifying the *correctness* of rules in Section 4.

3.3.2 Traceback

USB packets do not carry attribution data that can be used to determine the source or destination process of a packet. We therefore need to perform traceback to attribute packets to interfaces and processes.

Interfaces. As discussed in Section 2, a USB device can have multiple interfaces, each with a discrete functionality served by a device driver in the operating system. Once a driver is bound with an interface, it is able to communicate with that interface using USB packets.

Determining the driver responsible for receiving or sending a given USB packet is useful for precisely controlling device behaviors. However, identifying the responsible driver is not possible at the packet level, since the packets are already in transit and do not contain identifying information. While we could infer the responsible driver for simple USB devices, such as a mouse, this becomes unclear with composite USB devices with multiple interfaces (some of which may be served by the same driver).

To recover this important information from USB packets without changing each driver and extending the packet structure, we save the interface index into the kernel endpoint structure during USB enumeration.

This reverse mapping of interface to driver needs to be performed only once per device. The interface index distinguishes interfaces belonging to the same physical device and USB packets submitted by different driver instances. Once the mapping has been completed, the USB host controller is able to easily trace the originating interface back to the USB packets.

Processes. Similarly, tracking the destination or source process responsible for a USB packet is not trivial due to the way modern operating systems abstract device access from applications. For example, when communicating with USB storage devices, the operating system provides several abstractions between the application and the raw device, including a filesystem, block layer, and I/O scheduler. Furthermore, applications generally submit asynchronous I/O requests, causing the kernel to perform the communications task on a separate background thread.

This problem also appears when inspecting USB network device packets, including both wireline (e.g., Ethernet) dongles and wireless (e.g., WiFi) adapters. It is common for these USB device drivers to have their own RX/TX queues to boost the system performance using asynchronous I/O. In these cases, USB is an intermediate layer to encapsulate IP packets into USB packets for processing by the USB networking hardware.

These cases are problematic for USBFILTER because a naïve traceback approach will often only identify the kernel thread as the origin of a USB packet. To recover the process identifier (PID) of the true origin, we must ensure that this information persists between all layers within the operating system before the I/O request is transformed into a USB packet.¹

USBFILTER instruments the USB networking driver (*usbnet*), the USB wireless driver (*rt2x00usb*), the USB storage driver (*usb-storage*), as well as the block layer and I/O schedulers. Changes to the I/O schedulers are needed to avoid the potential merging of two block requests from different processes. By querying the rule database and USBFILTER modules, USBFILTER sets up a filter for all USB packets right before being dispatched to the devices.

3.3.3 Userspace Control

USBTABLES manages USBFILTER rules added in the kernel and saves all active rules in a database. Using udev, saved rules are flushed into the kernel automatically upon reboot. USBTABLES is also responsible for verifying the correctness of rules as we will discuss in Section 4. Once

¹USBFILTER does not overlap with Netfilter or any other IP packet filtering mechanisms which work along the TCP/IP stack.

verified, new rules will be synchronized with the kernel and saved locally.

If no user-defined rules are present, USBFILTER enforces default rules that are designed to prevent impact on normal kernel activities (e.g., USB hot-plugs). These rules can be overridden or augmented by the user as desired.

3.4 Deployment

We now demonstrate how we use existing security techniques in the deployment of USBFILTER. Attestation and MAC policy are necessary for providing complete mediation and tamperproof reference monitor guarantees, but not for the functionality of the system. The technologies we reference in this section are illustrative examples of how these goals can be met.

3.4.1 Platform Integrity

We deployed USBFILTER on a physical machine with a Trusted Platform Module (TPM). The TPM provides a root of trust that allows for a measured boot of the system and provides the basis for remote attestations to prove that the host machine is in a known hardware and software configuration. The BIOS’s core root of trust for measurement (CRTM) bootstraps a series of code measurements prior to the execution of each platform component. Once booted, the kernel then measures the code for user-space components (e.g., provenance recorder) before launching them using the Linux Integrity Measurement Architecture (IMA)[31]. The result is then extended into TPM PCRs, which forms a verifiable chain of trust that shows the integrity of the system via a digital signature over the measurements. A remote verifier can use this chain to determine the current state of the system using TPM attestation. Together with TPM, we also use Intel’s Trusted Boot (tboot)²

3.4.2 Runtime Integrity

After booting into the USBFILTER kernel, the runtime integrity of the TCB (defined in Section 3.1) must also be assured. To protect the runtime integrity of the kernel, we deploy a Mandatory Access Control (MAC) policy, as implemented by Linux Security Modules. We enable SELinux’s MLS policy, the security of which was formally modeled by Hicks et al. [20]. We also ensure that USBTABLES executes in a restricted environment and that the access to the rules database saved on the disk is protected by defining an SELinux Policy Module and compiling it into the SELinux Policy.

² See <http://sf.net/projects/tboot>

4 Security

In this section, we demonstrate that USBFILTER meets the security goals outlined in Section 3 using the deployment and configurations described in that section.

Complete Mediation (G1). As we previously discussed, USBFILTER must mediate all USB packets between devices and applications on the host. In order to ensure this, we have instrumented USBFILTER into the USB host controller, which is the last hop for USB packets before leaving the host machine and the first when entering it. Devices cannot initiate USB packet transmission without permission from the controller.

We also instrument the virtual USB host controller (*vhci*) to cover virtual USB devices (e.g., USB/IP). To support other non-traditional USB host controllers such as Wireless USB [19] and Media Agnostic USB [16], USBFILTER support is easily added via a simple kernel API call and the inclusion of a header file.

Tamperproof (G2). USBFILTER is statically compiled and linked into the kernel image to avoid being unloaded as a kernel module. The integrity of this runtime, the associated database, and user-space tools is assured through the SELinux policy as described in Section 3.4.2. Tampering with the kernel or booting a different kernel is the only way to bypass USBFILTER, and platform integrity measures provide detection capabilities for this scenario (Section 3.4.1).

Formal Verification (G3). The formal verification of USBFILTER rules is implemented as a logic engine within USBTABLES using GNU Prolog [11]. Instead of trying to prove that an abstract model of rule semantics is correctly implemented by the code, which is usually intractable for the Linux kernel, we limit our focus on rule correctness and consistency checking. Each time USBTABLES is invoked to add a new rule, the new rule and the existing rules are loaded into the logic engine for formal verification. This process only needs to be performed once when adding a new rule and USBFILTER continues to run while the verification takes place.

The verification checks for rules with the same conditions but different actions. These rules are considered conflicting and USBTABLES will terminate with error when this occurs. We define the correctness of a rule:

$$\begin{aligned} is_correct(R, \mathbb{R}) \leftarrow \\ & is_name_unique(R) \wedge \\ & are_condition_values_in_range(R) \wedge \\ & has_no_conflict_with_existing_rules(R, \mathbb{R}). \end{aligned}$$

where R is a new USBFILTER rule and \mathbb{R} for all other

existing rules maintained by USBFILTER. If the new rule has a unique name, all the values of conditions are in range, and it does not conflict with any existing rules, the rule is correct.

While the name and the value checks are straightforward, there are different conflicting cases between the conditions and the action, particularly when a rule does not contain all conditions. For example, a rule can be contradictory with, a sub rule of, or the same as another existing rule. As such, we define the general conflict between two rules as follows:

$$\begin{aligned} \text{general_conflict}(R_a, R_b) \leftarrow \\ \forall C_i \ni \mathcal{C}: \\ (\exists C_i^a \ni R_a \wedge \exists C_i^b \ni R_b \wedge \text{value}(C_i^a) \neq \text{value}(C_i^b)) \vee \\ (\exists C_i^a \ni R_a \wedge \nexists C_i^b \ni R_b) \vee \\ (\nexists C_i^a \ni R_a \wedge \exists C_i^b \ni R_b). \end{aligned}$$

A rule R_a is generally conflicted with another rule R_b if all conditions used by R_a are a subset of the ones specified in R_b . We consider a *general conflict* to occur if the new rule and an existing rule would fire on the same packet.

Based on the general conflict, we define *weak conflict* and *strong conflict* as follows:

$$\begin{aligned} \text{weak_conflict}(R_a, R_b) \leftarrow \\ \text{general_conflict}(R_a, R_b) \wedge \text{action}(R_a) = \text{action}(R_b). \\ \text{strong_conflict}(R_a, R_b) \leftarrow \\ \text{general_conflict}(R_a, R_b) \wedge \text{action}(R_a) \neq \text{action}(R_b). \end{aligned}$$

While weak conflict shows that the new rule could be a duplicate of an existing rule, strong conflict presents that this new rule would not work. The weak conflict, however, depending on the requirement and the implementation, may be allowed temporarily to shrink the scope of an existing rule while avoiding the time gap between the old rule removed and the new rule added. For instance, rule A drops any USB packets writing data into any external USB storage devices. Later on, the user decides to block write operations only for the Kingston thumb drive by writing rule B, which is weak conflicted with rule A, since both rules have the same destination and action. When the user wants to unblock the Kingston storage by writing rule C, rule C is strong conflicted with both rule A and B, since rule C has a different action, and will never work as expected because of rule A/B. By relying on the logic reasoning of Prolog, we are able to guarantee that a rule before added is formally verified no conflict with existing rules³.

³Note that all rules are monotonic by design, which means rules to be added cannot override existing ones. Future work will add general rules, which can be overwritten by new rules.

```

-d|--debug      enable debug mode
-c|--config    path to configuration file (TBD)
-h|--help       display this help message
-p|--dump       dump all the rules
-a|--add        add a new rule
-r|--remove     remove an existing rule
-s|--sync        synchronize rules with kernel
-e|--enable      enable usbfILTER
-q|--disable    disable usbfILTER
-b|--behave    change the default behavior
-o|--proc       process table rule
-v|--dev        device table rule
-k|--pkt        packet table rule
-l|--lum        LUM table rule
-t|--act        table rule action
-----
proc: pid,ppid,pgid,iuid,euid,gid,egid,comm
dev: busnum,devnum,portnum,ifnum,devpath,product,
     manufacturer,serial
pkt: types,direction,endpoint,address
lum: name
behavior/action: allow|drop

```

Figure 4: The output of “usbtables -h”. The permitted conditions are divided into 4 tables: the process table, the device table, the packet table, and the Linux USBFILTER Module (LUM) table.

Granular (G4). A USBFILTER rule can contain 21 different conditions, excluding the name and action field. We further divide these conditions into 4 tables, including the process, device, packet, and the Linux USBFILTER Module (LUM) table, as shown in Figure 4. The process table lists conditions specific to target applications; the device table contains details of USB devices in the system; the packet table includes important information about USB packets; and the LUM table determines the name of the LUM to be used if needed. Note that all LUMs should be loaded into the kernel before being used in USBFILTER rules.

Module Extension (G5). To support customized rule construction and deep USB packet analysis, USBFILTER allows system administrators to write Linux USBFILTER Modules (LUMs), and load them into the kernel as needed. To write a LUM, developers need only include the `<linux/usbfILTER.h>` header file in the kernel module, implement the callback `lum_filter_urb()`, and register the module using `usbfilter_register_lum()`. Once registered, the LUM can be referenced by its name in the construction of a rule. When a LUM is encountered in a rule, besides other condition checking, USBFILTER calls the `lum_filter_urb()` callback within this LUM, passing the USB packet as the sole parameter. The callback returns 1 if the packet matches the target of this LUM, 0 otherwise. Note that the current implementation supports only one LUM per rule.

5 Evaluation

The USBFILTER host machine is a Dell Optiplex 7010 with an Intel Quad-core 3.20 GHz CPU with 8 GB memory and is running Ubuntu Linux 14.04 LTS with kernel version 3.13. The machine has two USB 2.0 controllers and one USB 3.0 controller, provided by the Intel 7 Series/C210 Series chipset. To demonstrate the power of USBFILTER, we first examine different USB devices and provide practical use cases which are non-trivial for traditional access control mechanisms. Finally we measure the overhead introduced by USBFILTER.

The default behavior of USBFILTER in our host machine is to allow the USB packet if no rule matches the packet. A more constrained setting is to change the default behavior to drop, requiring each permitted USB device to need an allow rule. In this setting, malicious devices have to impersonate benign devices to allow communications, which are still regulated by the rules, e.g., no HID traffic allowed for a legit USB storage device. All tests use the same front-end USB 2.0 port on the machine.

5.1 Case Studies

Listen-only USB headset. The typical USB headset is a composite device with multiple interfaces including speakers, microphone, and volume control. Sensitive working environments may ban the use of USB headsets due to possible eavesdropping using the microphone [17]. Physically disabling the headset microphone is often the only mechanism for permanently removing it, as there is no other way to guarantee the microphone stays off. Users can mute or unmute the microphone using the desktop audio controls at any time after login. However, with USBFILTER, the system administrator can guarantee that the headset’s microphone remains disabled and cannot be enabled or accessed by users.

We use a Logitech H390 Headset to demonstrate how to achieve this guarantee on the USBFILTER host machine:

```
usbttables -a logitech-headset -v ifnum=2,product="Logitech USB Headset",manufacturer=Logitech -k direction=1 -t drop
```

This rule drops any incoming packets from the Logitech USB headset’s microphone. By adding the interface number (`ifnum=2`), we avoid breaking other functionality in the headset.

Customizing devices. To further show how USBFILTER can filter functionalities provided by USB devices, we use Teensy 3.2 [29] to create a complex USB device with five interfaces including a keyboard, a mouse, a joystick, and two serial ports. The keyboard contin-

ually types commands in the terminal, while the mouse continually moves the cursor. We can write USBFILTER rules to completely shutdown the keyboard and mouse functionalities:

```
usbttables -a teensy1 -v ifnum=2,manufacturer="Teensyduino",serial=1509380 -t drop  
usbttables -a teensy2 -v ifnum=3,manufacturer="Teensyduino",serial=1509380 -t drop
```

In these rules, we use condition “`manufacturer`” and “`serial`” (serial number) to limit the Teensy’s functionality. Different interface numbers represent the keyboard and the mouse respectively. After these rules applied, both the keyboard and the mouse return to normal.

Default-deny input devices. Next, we show how to defend against HID-based BadUSB attacks using USBFILTER. These types of devices are a type of *trojan horse*; they appear to be one device, such as a storage device, but secretly contain hidden input functionality (e.g., keyboard or mouse). When attached to a host, the device can send keystrokes to the host and perform actions as the current user.

First, we create a BadUSB storage device using a Rubber Ducky [18], which looks like a USB thumb drive but opens a terminal and injects keystrokes. Then we add following rules into the host machine:

```
usbttables -a mymouse -v busnum=1,devnum=4,portnum=2,  
    devpath=1.2,product="USB Optical Mouse",  
    manufacturer=PixArt -k types=1 -t allow  
usbttables -a mykeyboard -v busnum=1,devnum=3,  
    portnum=1,devpath=1.1,  
    product="Dell USB Entry Keyboard",  
    manufacturer=DELL -k types=1 -t allow  
usbttables -a noducky -k types=1 -t drop
```

The first two rules whitelist the existing keyboard and mouse on the host machine; the last rule drops any USB packets from other HID devices. After these rules are inserted into the kernel, reconnecting the malicious device does nothing. Attackers may try to impersonate the keyboard or mouse on the host machine. However, we have leveraged information about the physical interface (`busnum` and `portnum`) to write the first two rules, which would require the attacker to unplug the existing devices, plug the malicious device in, and impersonate the original devices including the device’s VID/PID and serial number. We leave authenticating individual USB devices to future work, however USBFILTER is extensible so that authentication can be added and used in rules.

Data exfiltration. To prevent data exfiltration from the host machine to USB storage devices, we write a LUM (Linux USBFILTER Module) to block the SCSI write command from the host to the device, as shown in Figure 9 in the Appendix. The LUM then registers itself with USBFILTER and can be referenced by its name in

rule constructions. In this case study, we use a Kingston DT 101 II 2G USB flash drive, and insert the following rule:

```
usbtables -a nodataexfil -v manufacturer=Kingston
    -l name=block_scsi_write -t drop
```

This rule prevents modification of files on the storage device. Interestingly, vim reports files on the device to be read-only, despite the filesystem reporting that the files are read-write. Since USBFILTER is able to trace packets back to the applications initiating I/O operations at the Linux kernel block layer, we are able to write rules blocking (or allowing) specific users or applications from writing to flash drive:

```
usbtables -a nodataexfil2 -o uid=1001
    -v manufacturer=Kingston
    -l name=block_scsi_write -t drop
usbtables -a nodataexfil3 -o comm=vim
    -v manufacturer=Kingston
    -l name=block_scsi_write -t drop
```

The first rule prevents the user with `uid=1001` from writing anything to the USB storage; the second blocks vim from writing to the storage. We can also block any writes to USB storage devices:

```
usbtables -a nodataexfil4
    -l name=block_scsi_write -t drop
```

USBFILTER logs dropped USB packets, and these logs can easily be used in a centralized alerting system, notifying administrators to unauthorized access attempts.

Webcam pinning. Webcams can easily be enabled and accessed by attackers from exploiting vulnerable applications. Once access has been established, the attacker can listen or watch the environment around the host computer. In this case study, we show how to use USBFILTER to restrict the use of a Logitech Webcam C310 to specific users and applications.

```
usbtables -a skype -o uid=1001,comm=skype -v
    serial=B4482A20 -t allow
usbtables -a nowebcam -v serial=B4482A20 -t drop
```

The serial number of the Logitech webcam is specified in the rules to differentiate any others that may be attached to the system as well as to prevent other webcams from being attached. The first rule allows USB communication with the webcam only if the user is `uid=1001` and the application is Skype. The following `nowebcam` rule drops other USB packets to the webcam otherwise. As expected, the user can use the webcam from his Skype but not from Pidgin, and other users cannot start video calls even with Skype.

USB charge-only. Another form of BadUSB attacks is DNS spoofing using smartphones. Once plugged into the host machine, the malicious phone automatically enables USB tethering, is recognized as a USB NIC by the host,

	Prolog Engine	Min	Avg	Med	Max	Dev
Time (20 rules)	128.0	239.8	288.0	329.0	73.2	
Time (100 rules)	132.0	251.7	298.0	485.0	78.6	

Table 1: Prolog reasoning time (μ s) averaged by 100 runs.

then injects spoofed DNS replies into the host. The resulting man-in-the-middle attack gives the attacker access to the host’s network communications without the authorization of the user. To prevent this attack, we use USBFILTER to prevent all USB packets from a Google Nexus 4 smartphone:

```
usbtables -a n4-charger -v product="Nexus 4" -t drop
```

This rule drops any USB packets to/from the phone, which enforces the phone as a pure charging device without any USB functionality. The phone is unable to be used for storage or tethering after the rule is applied.

We can construct a more specific charge-only rule:

```
usbtables -a charger -v busnum=1,portnum=4 -t drop
```

This rule specifies a specific physical port on the host and this port can only be used for charging. This type of rule is useful where USB ports may be exposed (e.g., on a point of sale terminal) and cannot be physically removed. It is also vital to defend against malicious devices whose firmware can be reprogrammed to forge the VID/PID such as BadUSB, since this type of rule only leverages the physical information on the host machine. USBFILTER can partition all physical USB ports and limit the USB traffic on each port.

5.2 Benchmarks

We first measure the performance of the user-space tool, USBTABLES. We then measure the overhead imposed by USBFILTER.

The measurement host is loaded with the rules mentioned in the case studies above before beginning benchmarking. When coupled with the default rules provided by USBFILTER, there are 20 total rules loaded in the kernel. We chose 20 because we believe that a typical enterprise host’s USB devices (e.g., keyboard, mouse, removable storage, webcam, etc.) will total less than 20. Then we load 100 rules in the kernel to understand the scalability of USBFILTER.

5.2.1 Microbenchmark

USBTABLES Performance. We measure the time used by the Prolog engine to formally verify a rule before it is added into the kernel. We loaded the kernel with 20 and

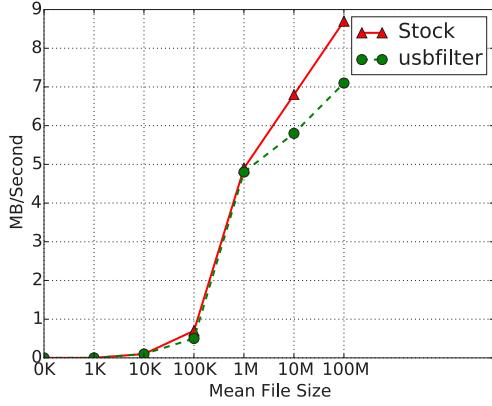


Figure 5: Filebench throughput (MB/s) using fileserver workload with different mean file sizes.

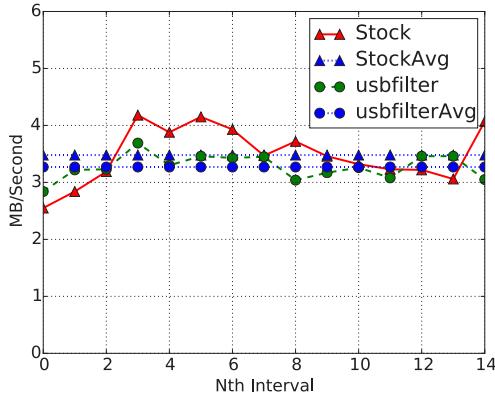


Figure 7: Iperf bandwidth (MB/s) using UDP with different time intervals.

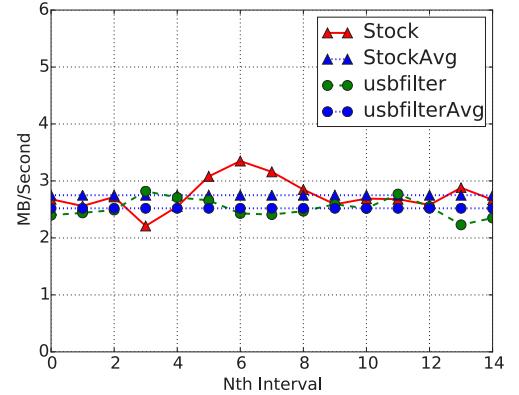


Figure 6: Iperf bandwidth (MB/s) using TCP with different time intervals.

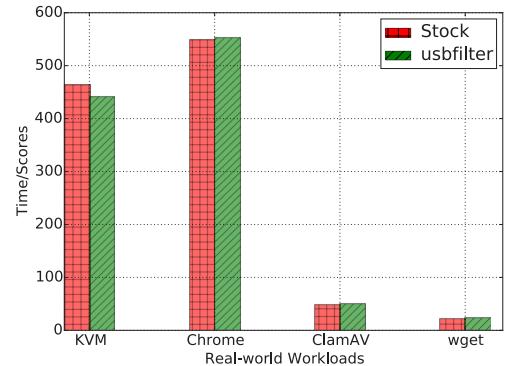


Figure 8: Performance comparison of real-world workloads.

Rule Adding	Min	Avg	Med	Max	Dev
Time (20 rules)	5.1	5.9	6.1	6.6	0.3
Time (100 rules)	4.9	5.9	6.1	6.8	0.4

Table 2: Rule adding operation time (ms) averaged by 100 runs.

100 rules and measured the time to process the rules. For each new rule, the Prolog engine needs to go through the existing rules and check for conflicts.

We measured 100 trials of each test. The performance of the Prolog engine is shown in Table 1. The average time used by the Prolog engine is $239.8 \mu s$ with 20 rules and $251.7 \mu s$ with 100 rules. This fast speed is the result of using GNU Prolog (`gplc`) compiler to compile Prolog into assembly for acceleration. We also measure the overhead for USBTABLES to add a new rule to the kernel space. This includes loading existing rules into the Prolog engine, checking for conflicts, saving the rule

USB Enumeration	Min	Avg	Med	Max	Dev	Cost
Stock Kernel	32.0	33.9	34.1	34.8	0.6	N/A
USBFILTER (20 rules)	33.2	34.4	34.3	35.8	0.7	1.5%
USBFILTER (100 rules)	33.9	34.8	34.6	36.0	0.5	2.7%

Table 3: USB enumeration time (ms) averaged by 20 runs.

locally, passing the rule to the kernel, and waiting for the acknowledgment. As shown in Table 2, the average time of adding a rule using USBTABLES stays at around 6 ms in both cases, which is a negligible one-time cost.

USB Enumeration Overhead. For this test, we used the Logitech H390 USB headset, which has 4 interfaces. We manually plugged the headset into the host 20 times. We then compare the results between the USBFILTER kernel with varying numbers of rules loaded and the stock Ubuntu kernel, where USBFILTER is fully disabled,

Packet Filtering	Min	Avg	Med	Max	Dev
Time (20 rules)	2.0	2.6	3.0	5.0	0.5
Time (100 rules)	2.0	9.7	10.0	15.0	1.0

Table 4: Packet filtering time (μ s) averaged by 1500 packets.

Configuration	1K	10K	100K	1M	10M	100M
Stock	97.6	98.1	99.2	105.5	741.7	5177.7
USBFILTER	97.7	98.2	99.6	106.3	851.5	6088.4
Overhead	0.1%	0.1%	0.4%	0.8%	14.8%	17.6%

Table 5: Latency (ms) of the `fileserver` workload with different mean file sizes.

as shown in Table 3. The average USB enumeration time is 33.9 ms for the stock kernel and 34.4 ms and 34.8 ms for the USBFILTER kernel with 20 and 100 rules preloaded respectively. Comparing to the stock kernel, USBFILTER only introduces 1.5% and 2.7% overheads, or less than 1 ms even with 100 rules preloaded.

Packing Filtering Overhead. The overhead of USB enumeration introduced by USBFILTER is the result of packet filtering and processing performed on each USB packet, since there may be hundreds of packets during USB enumeration, depending on the number of interface and endpoints of the device. To capture this packet filtering overhead, we plug in a Logitech M105 USB Optical Mouse, and move it around to generate enough USB packets. We then measure the time used by USBFILTER to determine whether the packet should be filtered/dropped or not for 1500 packets, as shown in Table 4. The average cost per packet are 2.6 μ s and 9.7 μ s respectively, including the time to traverse all the 20/100 rules in the kernel, and the time used by the benchmark itself to get the timing and print the result. The 100-rule case shows that the overhead of USBFILTER is quadrupled when the number of rule increases by one order of magnitude. As we mentioned before, most common USB usages could be covered within 20 rules. We assume it is rare for a system to have 100 rules for different USB devices. To search in hundreds of rules efficiently, we can setup a hash table using e.g., USB port numbers as keys to save rules instead of a linear array (list) currently implemented.

5.2.2 Macrobenchmark

We use filebench [37] and iperf [42] to measure throughputs and latencies of file operations, and bandwidths of network activities, under the stock kernel and the USBFILTER kernel, using different USB devices. The

USBFILTER kernel is loaded with 20 rules introduced in the case studies before benchmarking.

Filebench. We choose the `fileserver` workload in filebench, with the following settings: the number of files in operation is 20; the number of working threads is 1; the run time for each test case is 2 minutes; the mean file size in operation ranges from 1 KB to 100 MB; all other settings are default provided by filebench. These settings emulate a typical usage of USB storage devices, where users plug in flash drives to copy or edit some files. All file operations happen in a SanDisk Cruzer Fit 16 GB flash drive. The throughputs under the stock kernel and the USBFILTER kernel are demonstrated in Figure 5. When the mean file size is less than 1 MB, the throughput of USBFILTER is close to the one of the stock kernel. Since there is at most 20×1 MB data involved in block I/O operations, both the stock kernel and USBFILTER can handle this data size smoothly. When the mean file size is greater than 1 MB, USBFILTER shows lower throughputs comparing to the stock kernel, as the result of rule matching for each USB packet. Compared to the stock kernel, USBFILTER imposes 14.7% and 18.4% overheads when the mean file sizes are 10 MB and 100 MB respectively. That is, when there is 20×100 MB (2 GB) involved in block I/O operations, the throughput decreases from 8.7 MB/s to 7.1 MB/s, when USBFILTER is enabled.

The corresponding latencies are shown in Table 5. The latency of USBFILTER is higher than the stock kernel. Following the throughput model, the latencies between the two kernels are close when the mean file size is less than 1 MB. The overhead introduced by USBFILTER is less than 1.0%. When the mean file sizes are 10 MB and 100 MB, USBFILTER imposed 14.8% and 17.6% overheads in latency, comparing to the stock kernel. That is, to deal with 20×100 MB data, users need one more second to finish all the operations with USBFILTER enabled, which is acceptable for most users.

iperf. We use iperf to measure bandwidths of upstream TCP and UDP communications, where the host machine acts as a server, providing local network access via a Ralink RT5372 300 Mbps USB wireless adapter. The time interval for each transmission is 10 seconds, and each test runs 5 minutes (30 intervals). For TCP, we use the default TCP window size 64 KB; for UDP, we use the default available UDP bandwidth size 10 MB. The TCP bandwidths of the two kernels are shown in Figure 6, where we aggregate each two intervals into one, reducing the number of sampling points from 30 to 15, and the average bandwidths are also listed in dot lines. Though having different transmission patterns, the average bandwidths of both are close, with the stock kernel at 2.75 Mbps and USBFILTER at 2.52 Mbps. Comparing to

the stock kernel, USBFILTER introduces 8.4% overhead.

The UDP benchmarking result closely resembles TCP, as shown in Figure 7. Regardless of transmission patterns, average bandwidth of the two kernels is similar, with the stock kernel at 3.48 Mbps and USBFILTER at 3.27 Mbps. Comparing to the TCP transmission, UDP transmission is faster due to the simpler design/implementation of UDP, and USBFILTER introduces 6.0% overhead. In both cases, USBFILTER has demonstrated a low impact to the original networking component.

5.3 Real-world Workloads

To better understand the performance impact of USBFILTER, we generate a series of real-world workloads to measure typical USB use cases. In the KVM [24] workload, we create and install a KVM virtual machine automatically from the Ubuntu 14.04 ISO image file (581 MB) saved on USB storage. In the Chrome workload, we access the web browser benchmark site [5] via a USB wireless adapter. In the ClamAV [25] workload, we scan the unzipped Ubuntu 14.04 ISO image saved on the USB storage for virus using ClamAV. In the wget workload, we download the Linux kernel 4.4 (83 MB) via the USB wireless adapter using wget. The USB storage is the SanDisk 16 GB flash drive, and the USB wireless adapter is the Ralink 300 Mbps wireless card. All time measurements are in seconds except the Chrome workload, where scores are given, and are divided by 10 to fit into the figure. Figure 8 shows the comparison between the two kernels when running these workloads. In all workloads, USBFILTER either performs slightly better than the stock kernel, or imposes a small overhead compared to the stock kernel in our test. It is clear that USBFILTER approximates the original system performance.

5.4 Summary

In this section, we showed how USBFILTER can help administrators prevent access to unauthorized (and unknown) device interfaces, restrict access to authorized devices using application pinning, and prevent data exfiltration. Our system introduces between 3 and 10 μ s of latency on USB packets while checking rules, introducing minimal overhead on the USB stack.

6 Discussion

6.1 Process Table

We have successfully traced each USB packet to its originating application for USB storage devices by passing the PID information along the software stack from the

VFS layer, through the block layer, to the USB layer within the kernel. However, it is not always possible to find the PID for each USB packet received by the USB host controller. One example is HID devices, such as keyboards and mouses. Keystrokes and mouse movements happen in the interrupt (IRQ) context, where the current stopped process has nothing to do with this USB packet. All these packets are delivered to the Xorg server in the user space, which then dispatches the inputs to different applications registered for different events. USBFILTER is able to make sure that only Xorg can receive inputs from the keyboard and mouse. To guarantee the USB packet delivered to the desired application, we can enhance the Xorg server to understand USBFILTER rules.

The other example comes from USB networking devices. Though we have enhanced the general USB wire-line driver `usbnet` to pass the PID information into each USB packet, unlike USB storage devices sharing the same `usb-storage` driver, many USB Ethernet dongles have their own drivers instead of using the general one. Even worse, there is no general USB wireless driver at all. Depending of the device type and model, one may need to instrument the corresponding driver to have the PID information, like what we did for `rt2800usb` driver. Future work will introduce a new USB networking driver framework to be shared by specific drivers, providing a unified interface for passing PID information into USB packets.

Another issue of using process table in USBFILTER rules is TOCTTOU (time-of-check-to-time-of-use) attacks. A malicious process can submit a USB packet to the kernel and exit. When the packet is finally handled by the host controller, USBFILTER is no longer able to find the corresponding process given the PID. Fortunately, these attacks does not impact rules without process tables. When process information is crucial to the system, we recommending using `USBTABLES` to change the default behavior to “drop”, make sure that no packet would get through without an explicit matching rule.

6.2 System Caching

USBFILTER is able to completely shut down any write operations to external USB storage devices, preventing any form of data exfiltration from the host machine. Similarly, one can also write a “`block_scsi_read`” LUM to stop read operations from storage devices. Nevertheless, this LUM may not be desired or work as expected in reality. To correctly mount the filesystem in the storage device, the kernel has to read the metadata saved in the storage. One solution would be to delay the read blocking till the filesystem is mounted. However, for performance considerations, the Linux kernel also reads ahead some data in the storage, and brings it into the system

cache (page cache). All following I/O operations will happen in the memory rather than the storage. While memory protection is out of scope for this paper, we rely on the integrity of the kernel to enforce the MAC model it applies. Write operations, even though in the memory, will be flushed into the storage, where USBFILTER is able to provide a strong and useful guarantee.

6.3 Packet Analysis From USB Devices

Because of the master-slave nature of the USB protocol, we do not setup USBFILTER in the response path, which is from the device to the host, due to performance considerations. However, enabling USBFILTER in the response path provides new opportunities to defend against malicious devices and users, since the response packet could be inspected with the help of USBFILTER. For example, one can write a LUM to limit the capability of a HID device, such as allowing only three different key actions from a headset’s volume control button, which is implemented by GoodUSB as a customized keyboard driver, or disabling sudo commands for unknown keyboards. Another useful case is to filter the spoofing DNS reply message embedded in the USB packet sent by malicious smart phones or network adapters, to defend against DNS cache poisoning. We are planning to investigate these new case studies in future work.

6.4 Malicious USB Drivers and USB Covert Channels

While BadUSB is the most prominent attack that exploits the USB protocol, we observe that using USB communication as a side channel to steal data from host machines, or to inject malicious code into hosts, is another technically mature and plausible threat. On the Linux platform, with the development of libusb [14], more USB drivers run within user space and can be delivered as binaries. On Windows platform, PE has been a common format of device drivers. To use these devices, users have to run these binary files without knowing if these drivers are doing something else in the meantime.⁴ For instance, USB storage devices should use bulk packets to transfer data per the USB spec. However, a malicious storage driver may use control packets to stealthily exfiltrate data as long as the malicious storage is able to decode the packet. This works because control transfers are mainly used during the USB enumeration process. With the help of USBFILTER, one can examine each USB packet, and filter unrecognized ones without breaking the normal functionality of the device.

⁴N.B. that there are ways to instrument DLL files on Windows platform, though this does not appear to be commonly done with drivers.

6.5 Usability Issues

To write USBFILTER rules, one needs some knowledge about the USB protocol in general, as well as the target USB device. The `lsusb` command under Linux provides a lot of useful information that can directly be mapped into rule construction. Another tool `usb-devices` also helps users understand USB devices. Windows has a GUI program `USBView` to visualize the hierarchy and configuration of USB devices plugged into the host machine. While users can write some simple rules, we expect that developers will provide useful LUMs, which may require deep understanding of the USB protocol and domain specific knowledge (e.g., SCSI, and will share these LUMs with the community. We will also provide more useful LUMs in the future.

7 Related Work

Modern operating systems implicitly approve all interfaces on any device that has been physically attached to the host. Due to this, a wide range of attacks have been built on USB including malware and data exfiltration on removable storage [15, 34, 46], tampered device firmware [27, 7], and unauthorized devices [1]. These attacks fall into two major categories: those that involve data ingress and egress via removable storage and those that involve the attachment of unknown USB interfaces.

Proposals for applying access control to USB storage devices [12, 28, 38, 48] fall short because they cannot guarantee that the USB write requests are blocked from reaching the device. Likewise, defenses against unauthorized or malicious device interfaces [41, 33] and disabling device drivers are coarse and cannot distinguish between desired and undesired usage of a particular interface. Another solution employed by the Windows Embedded platform [26] binds USB port numbers with the VID/PID/CID (device class ID) information of devices to accept/reject the device plugged in. While CID helps limit the usage of the device, this solution does not work for composite devices equipped with multiple interfaces (with different CIDs). Besides, users may have to update the policy each time when different devices are plugged into the same port. Given the increasing ubiquity of USB, this is not a sustainable solution. Guardat demonstrates a means of expressing a robust set of rules for storage access but requires substantial new mechanisms for operation within a host computer, such as implementation within a hybrid disk microcontroller [45].

Netfilter [40] has become the de facto network firewall standard on Linux due to its ability to perform fine-grained filtering on network packets between applications and the physical network interface. Netfilter

can prevent compromise of a program by preventing unwanted packets from reaching the process. Similarly, our system can defend processes by denying USB traffic before it reaches its destination.

Furthermore, fine-grained filtering has been applied to the usage of filesystem objects by applications [13, 35], however, these filters take place *after* the host and operating system have enumerated the device and loaded any device drivers. USBFILTER applies filtering at the USB packet layer, preventing unauthorized access to interfaces regardless of whether they have been approved elsewhere. Since our system operates between the device drivers and the USB host controller and traces packets back to their source or destination application, USBFILTER can uniquely filter access to any USB interface.

While USBFILTER working in the host operating system directly, other USB security solutions make use of virtualization. GoodUSB [41] leverages a QEMU-KVM as a honeypot to analyze malicious USB devices, while Cinch [3] separates the trusted USB host controller and untrusted USB devices into two QEMU-KVMs, between which a gateway is used to apply policies on USB packets. By mitigating the need for additional components for standard operation, we believe that USBFILTER is better suited for adoption within operating system kernels.

USBFILTER protects the host machine from malicious USB devices, but there are solutions as well for exploring the protection of devices from malicious hosts. USB fingerprinting [6] establishes the host machine identity using USB devices, while Kells [8] protects the USB storage device by attesting the host machine integrity.

Wang and Stavrou [47] suggest that a “USB firewall” might protect against exploitation attacks but do not discuss the complexities of how such a mechanism could be designed or implemented.

8 Conclusion

USB attacks rely on hosts automatically authorizing any physically-attached device. Attackers can discreetly connect unknown and unauthorized interfaces, causing device drivers to be automatically loaded and allowing malicious devices access to the host. In this paper, we prevent unauthorized devices from accessing a host with USBFILTER, the first packet-level access control system for USB. Through tracing each packet back to its associated process, our system can successfully block unauthorized interfaces and restrict access to devices by process. With a default deny policy for new devices, administrators can restrict connection of unknown devices using granular identifiers such as serial number. Our experiments test USBFILTER using a range of I/O benchmarks and find that it introduces minimal overhead. The result is a host that is unresponsive to attacks that may try

to discreetly introduce unknown functionality via USB while maintaining high performance.

Acknowledgements

This work is supported in part by the US National Science Foundation under grant numbers CNS-1540217, CNS-1540218 and CNS-1464088.

References

- [1] TURNIPSCHOOL - NSA playset. <http://www.nsaplayset.org/turnipschool>.
- [2] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, 1972.
- [3] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walish. Defending against malicious peripherals. *arXiv preprint arXiv:1506.01449*, 2015.
- [4] J. Bang, B. Yoo, and S. Lee. Secure usb bypassing tool. *digital investigation*, 7:S114–S120, 2010.
- [5] Basemark, Inc. Basemark browsermark. <http://web.basemark.com/>, 2015.
- [6] A. Bates, R. Leonard, H. Pruse, K. R. B. Butler, and D. Lowd. Leveraging USB to Establish Host Identity Using Commodity Devices. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, NDSS ’14, February 2014.
- [7] M. Brocker and S. Checkoway. iseeyou: Disabling the macbook webcam indicator led. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 337–352, 2014.
- [8] K. R. B. Butler, S. E. McLaughlin, and P. D. McDaniel. Kells: a protection framework for portable data. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 231–240. ACM, 2010.
- [9] A. Caudill and B. Wilson. Phison 2251-03 (2303) Custom Firmware & Existing Firmware Patches (BadUSB). *GitHub*, 26, Sept. 2014.
- [10] Compaq, Hewlett-Packard, Intel, Microsoft, NEC, and Phillips. Universal Serial Bus Specification, Revision 2.0, April 2000.
- [11] D. Diaz et al. The GNU Prolog web site. <http://gprolog.org/>.
- [12] S. A. Diwan, S. Perumal, and A. J. Fatah. Complete security package for USB thumb drive. *Computer Engineering and Intelligent Systems*, 5(8):30–37, 2014.
- [13] W. Enck, P. McDaniel, and T. Jaeger. PinUP: Pinning user files to known applications. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 55–64. ieeexplore.ieee.org, Dec. 2008.
- [14] J. Erdfelt and D. Drake. Libusb homepage. *Online*, <http://www.libusb.org>.
- [15] N. Falliere, L. O. Murchu, and E. Chien. W32. Stuxnet Dossier. 2011.
- [16] U. I. Forum. Media Agnostic Universal Serial Bus Specification, Release 1.0a, July 2015.
- [17] D. Genkin, A. Shamir, and E. Tromer. RSA key extraction via Low-Bandwidth acoustic cryptanalysis. In *Advances in Cryptology – CRYPTO 2014*, Lecture Notes in Computer Science, pages 444–461. Springer Berlin Heidelberg, 17 Aug. 2014.

- [18] Hak5. Episode 709: USB Rubber Ducky Part 1. <http://hak5.org/episodes/episode-709>, 2013.
- [19] Hewlett-Packard, Intel, LSI, Microsoft, NEC, Samsung, and ST-Ericsson. Wireless Universal Serial Bus Specification 1.1, September 2010.
- [20] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A Logical Specification and Analysis for SELinux MLS Policy. *ACM Trans. Inf. Syst. Secur.*, 13(3):26:1–26:31, July 2010.
- [21] T. Hirofuchi, E. Kawai, K. Fujikawa, and H. Sunahara. USB/IP-A peripheral bus extension for device sharing over IP network. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 42–42, 2005.
- [22] IronKey, Inc. Access Enterprise. <http://www.ironkey.com/en-US/access-enterprise/>, 2015.
- [23] Jeremy Moskowitz. Managing hardware restrictions via group policy. <https://technet.microsoft.com/en-us/magazine/2007.06.grouppolicy.aspx>, 2007.
- [24] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.
- [25] T. Kojm. Clamav, 2004.
- [26] Microsoft Windows Embedded 8.1 Industry. Usb filter (industry 8.1). [https://msdn.microsoft.com/en-us/library/dn449350\(v=winembedded.82\).aspx](https://msdn.microsoft.com/en-us/library/dn449350(v=winembedded.82).aspx), 2014.
- [27] K. Nohl and J. Lell. BadUSB—On accessories that turn evil. *Black Hat USA*, 2014.
- [28] D. V. Pham, M. N. Halgamuge, A. Syed, and P. Mendis. Optimizing Windows Security Features to Block Malware and Hack Tools on USB Storage Devices. In *Progress in Electromagnetics Research Symposium*, 2010.
- [29] PJRC. Teensy 3.1. <https://www.pjrc.com/teensy/teensy31.html>, 2013.
- [30] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.
- [31] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. *Proceedings of the 13th USENIX Security Symposium*, 2004.
- [32] SANS Institute. Real World ARP Spoofing. <http://pen-testing.sans.org/resources/papers/gcih/real-world-arp-spoofing-105411>, 2003.
- [33] S. Schumilo, R. Spenneberg, and H. Schwartke. Don’t trust your USB! How to find bugs in USB device drivers. In *Blackhat Europe*, Oct. 2014.
- [34] S. Shin and G. Gu. Conficker and Beyond: A Large-scale Empirical Study. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC ’10, pages 151–160, New York, NY, USA, 2010. ACM.
- [35] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1:43, 2001.
- [36] J. Stewart. Dns cache poisoning—the next generation, 2003.
- [37] Sun Microsystems, Inc. and FSL at Stony Brook University. Filebench. http://filebench.sourceforge.net/wiki/index.php/Main_Page, 2011.
- [38] A. Tetmeyer and H. Saidian. Security Threats and Mitigating Risk for USB Devices. *Technology and Society Magazine, IEEE*, 29(4):44–49, winter 2010.
- [39] The Information Assurance Mission at NSA. Defense against Malware on Removable Media. https://www.nsa.gov/ia/_files/factsheets/mitigation_monday_3.pdf, 2007.
- [40] The Netfilter Core Team. The Netfilter Project: Packet Mangling for Linux 2.4. <http://www.netfilter.org/>, 1999.
- [41] D. J. Tian, A. Bates, and K. Butler. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 261–270, New York, NY, USA, 2015. ACM.
- [42] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The tcp/udp bandwidth measurement tool. <http://dast.nlanr.net/Projects>, 2005.
- [43] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey. Users Really Do Plug in USB Drives They Find. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P ’16)*, San Jose, California, USA, May 2016.
- [44] USB Implementers Forum. USB-IF statement regarding USB security. http://www.usb.org/press/USB-IF_Statement_on_USB_Security_FINAL.pdf.
- [45] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the Tenth European Conference on Computer Systems*, page 13. ACM, 2015.
- [46] J. Walter. "Flame Attacks": Briefing and Indicators of Compromise. *McAfee Labs Report*, May 2012.
- [47] Z. Wang and A. Stavrou. Exploiting Smart-phone USB Connectivity for Fun and Profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC ’10, 2010.
- [48] B. Yang, D. Feng, Y. Qin, Y. Zhang, and W. Wang. TMSUI: A Trust Management Scheme of USB Storage Devices for Industrial Control Systems. Cryptology ePrint Archive, Report 2015/022, 2015. <http://eprint.iacr.org/>.

Appendix

```
1  /*
2   * lbsw - A LUM kernel module
3   * used to block SCSI write command within USB packets
4   */
5 #include <linux/module.h>
6 #include <linux/usbfilter.h>
7 #include <scsi/scsi.h>
8
9 #define LUM_NAME           "block_scsi_write"
10 #define LUM_SCSI_CMD_IDX   15
11
12 static struct usbfilter_lum lbsw;
13 static int lum_registered;
14
15 /*
16  * Define the filter function
17  * Return 1 if this is the target packet
18  * Otherwise 0
19  */
20 int lbsw_filter_urb(struct urb *urb)
21 {
22     char opcode;
23
24     /* Has to be an OUT packet */
25     if (usb_pipein(urb->pipe))
26         return 0;
27
28     /* Make sure the packet is large enough */
29     if (urb->transfer_buffer_length <= LUM_SCSI_CMD_IDX)
30         return 0;
31
32     /* Make sure the packet is not empty */
33     if (!urb->transfer_buffer)
34         return 0;
35
36     /* Get the SCSI cmd opcode */
37     opcode = ((char *)urb->transfer_buffer)[LUM_SCSI_CMD_IDX];
38
39     /* Current only handle WRITE_10 for Kingston */
40     switch (opcode) {
41     case WRITE_10:
42         return 1;
43     default:
44         break;
45     }
46
47     return 0;
48 }
49
50 static int __init lbsw_init(void)
51 {
52     pr_info("lbsw: Entering: %s\n", __func__);
53     snprintf(lbsw.name, USBFILTER_LUM_NAME_LEN, "%s", LUM_NAME);
54     lbsw.lum_filter_urb = lbsw_filter_urb;
55
56     /* Register this lum */
57     if (usbfilter_register_lum(&lbsw))
58         pr_err("lbsw: registering lum failed\n");
59     else
60         lum_registered = 1;
61
62     return 0;
63 }
64
65 static void __exit lbsw_exit(void)
66 {
67     pr_info("exiting lbsw module\n");
68     if (lum_registered)
69         usbfilter_deregister_lum(&lbsw);
70 }
71
72 module_init(lbsw_init);
73 module_exit(lbsw_exit);
74
75 MODULE_LICENSE("GPL");
76 MODULE_DESCRIPTION("lbsw module");
77 MODULE_AUTHOR("dtrump");
```

Figure 9: An example Linux USBFILTER Module that blocks writes to USB removable storage.

Micro-Virtualization Memory Tracing to Detect and Prevent Spraying Attacks

Stefano Cristalli

Università degli Studi di Milano

Mattia Pagnozzi

Università degli studi di Milano

Mariano Graziano

Cisco Systems Inc.

Andrea Lanzi

Università degli Studi di Milano

Davide Balzarotti

Eurecom

Abstract

Spraying is a common payload delivery technique used by attackers to execute arbitrary code in presence of Address Space Layout Randomisation (ASLR). In this paper we present *Graffiti*, an efficient hypervisor-based memory analysis framework for the detection and prevention of spraying attacks. Compared with previous solutions, our system is the first to offer an efficient, complete, extensible, and OS independent protection against all spraying techniques known to date. We developed a prototype open source framework based on our approach, and we thoroughly evaluated it against all known variations of spraying attacks on two operating systems: Linux and Microsoft Windows. Our tool can be applied out of the box to protect any application, and its overhead can be tuned according to the application behavior and to the desired level of protection.

1 Introduction

Memory corruption vulnerabilities are currently one of the biggest threat to software and information security. Education plays a very important role in this area, making programmers aware of common threats and teaching them how to avoid mistakes that may lead to exploitable bugs in their code. However, education alone is not enough, and a good defense in depth approach requires also to put in place multiple layers of mitigation, detection, and exploit prevention mechanisms.

In this field, over the past decade we have witnessed a constant arms race, with the system designers of compilers and operating systems on one side, and the attackers on the other. Over the years, the former have introduced many new security features to increase the complexity of exploiting memory corruption vulnerabilities [31, 6, 41, 9, 40]. This list includes stack canaries [13, 12], data execution prevention (DEP) [2], Address Space Layout Randomization (ASLR) [43, 7, 26, 8], Structured Ex-

ception Handling Overwrite Protection (SEHOP) [29], and Control Flow Integrity [3]—just to name some of the most popular solutions. Even though the combination of all these techniques have certainly increased the security of modern operating systems, no matter how high the bar was set, attackers have always found a way to overcome it to take control of a vulnerable system.

ASLR is certainly one of the most common and successful techniques adopted by modern operating systems. In fact, the objective of the majority of memory corruption exploits is to allow the attacker to execute arbitrary code in the context of a vulnerable process. The code can be injected by the attacker herself, or it can be constructed by reusing instructions already present in memory (e.g., in the case of return-to-libc or return oriented programming). Either way, the attacker needs to know where such code is located in memory, in order to divert the control flow of the application to that precise address. And here is where ASLR plays its role: by completely randomizing the layout of the process memory, it makes much harder for the attacker to predict where a certain buffer (or an existing code gadget) will be located at run-time. Unfortunately, attackers found a very simple and effective solution to overcome this protection: fill the memory with tens of thousands of identical copies of the same malicious code, and then jump to a random page¹, hoping to land in one of the pre-loaded areas. This makes this payload delivery technique, called *spraying*, one of the key elements used in most of the recent memory corruption exploits.

Researchers have been looking for ways to mitigate this technique. Unfortunately, the few solutions proposed so far [36, 16, 21] were all tailored to defend 1) a particular application (typically the JavaScript interpreter in Internet Explorer), 2) using a given memory allocator, 3) in a specific operating system, and 4) against

¹Often a fixed address located on the process heap.

a single form of heap spraying. This made these solutions difficult to port to other environments, and unable to cope with all possible variations of heap spraying attacks. In fact, the original heap spraying attack is now just the tip of the iceberg. The technique has rapidly evolved in different directions, for example by taking advantage of just in time compilers (JIT), by focusing on the allocation of pools in the OS kernel, or by relying on stack pivoting to spray data instead of code. We strongly believe that the increased adoption and sophistication of heap spraying techniques clearly demonstrate the need for a general and comprehensive solution to this problem.

In this paper we present Graffiti, a hypervisor-based solution for the detection and prevention of all known variations of spraying attacks. We decided to implement our solution at the hypervisor level to obtain the first *OS-independent, allocator-agnostic* approach to track memory allocations that does not depend on the knowledge of the protected process, or system. By leveraging a novel micro-virtualization technique, Graffiti proposes an efficient and OS-agnostic framework to monitor memory allocations of arbitrary applications. The system is modular, and relies on a set of plug-ins to detect suspicious patterns in memory in realtime. For example, we developed a set of different detection modules based on statistical inference, designed to precisely identify all known spectrum of spraying attacks known to date. Moreover, while all the previous techniques [36, 21] focused on the defense of a particular application or memory allocator against a single form of heap spraying, our system offers the first general and portable solution to the problem.

Graffiti also offers a hot-plugging capability and therefore it can be installed on-the-fly without rebooting the machine and without modifying the native operating system. Our experiments, conducted both on Linux and Microsoft Windows, show that Graffiti has no false negatives and low false positives, with an overhead similar to the one of previous, much more limited, solutions.

In summary, our work makes the following contributions:

- We present the principles, design, and implementation of an effective real-time memory analysis framework. On top of our framework, we developed a set of heuristics to detect existing heap spraying techniques. To the best of our knowledge, we are the first to present a general, efficient, and comprehensive framework that can be applied to all modern operating systems and all existing applications.
- We propose a novel micro-virtualization technique that allows Graffiti to monitor the entire system in

terms of both processes and kernel threads, with low overhead.

- We have developed a prototype tool, and performed an experimental evaluation on several existing real-world spraying techniques. Our experiments show that the system is able to detect all the classes of spraying attacks we analyzed with low false positives and acceptable performance.
- We released the source code of the current Graffiti prototype, which is available at the following link: <https://github.com/graffiti-hypervisor/graffiti-hypervisor>

The rest of the paper is organized as follows. Section 2 provides background information on spraying attacks. Section 3 provides preliminary notions about Intel VT-x technology. Section 4, Section 5, Section 6 describe our solution from an architectural point of view. Section 7 reports results on evaluating Graffiti. Section 8 discusses about the security evaluation of our system. Section 9 compares our work with other relevant research and Section 10 discusses future directions and concludes the paper.

2 Spraying Attacks

Heap spraying is a payload delivery technique that was publicly used for the first time in 2001 in the telnetd remote root exploit [44] and in the eEye’s ISS AD20010618 exploit [15]. The technique became popular in 2004 as a way to circumvent Address Space Layout Randomization (ASLR) in a number of exploits against Internet Explorer [46, 47, 38].

Since 2004, spraying attacks have evolved and became more reliable thanks to improvements proposed by Sotirov [42] and Daniel [14] for a precise heap manipulation. Spraying can now be classified in two main categories, based on the protection mechanisms in place in the target machine: *Code Spraying* and *Data Spraying*. If Data Execution Prevention [2] (DEP) is not enabled, the attacker can perform the exploit by directly spraying the malicious code (e.g., the shellcode) in the victim process memory. On the other hand, when the system uses the DEP protection, the attacker would not be able to execute the injected code. To overcome this problem, two main approaches have been proposed: (a) perform the heap spraying by taking advantage of components that are not subjected to DEP, such as Just in Time Compilers (JITs), or (b) inject plain data that points to Return Oriented Programming (ROP) gadgets. While the internal details between the three aforementioned approaches

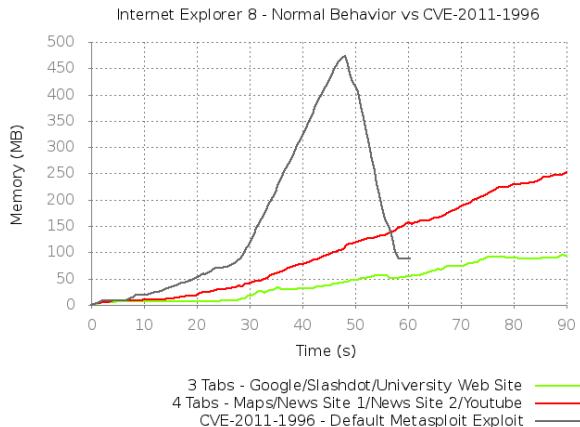


Figure 1: Heap Spraying attack

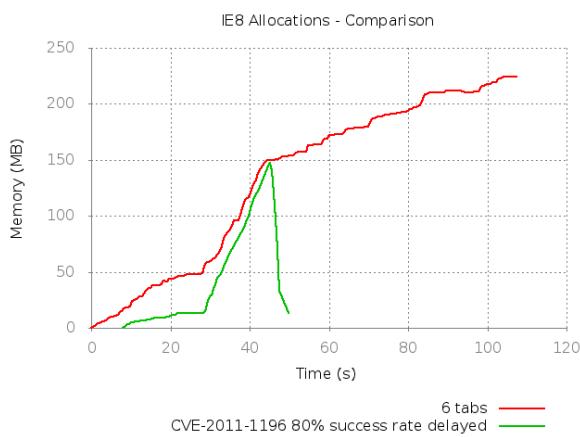


Figure 2: Mimicry attack

may be quite different, what is important for our research is that all these techniques share the same goal, i.e., to control the target dynamic memory allocation in order to obtain a memory layout that allows arbitrary code execution in a reliable way.

It is important to note that spraying is still a valuable technique also in x86_64-based operating systems. In particular, this is the case for user-after-free vulnerabilities – but spraying can also be used in conjunction with vulnerabilities in the ASLR implementation [10], in particular types of vulnerabilities [20], or because of the wide adoption of 32bit processes in 64bit operating systems (as recently shown by Skylined [39]).

2.1 Memory Footprint

The first characteristic of a heap spraying attack that comes to mind is the large amount of memory that is suddenly allocated by a process. Therefore we could erroneously believe that this unusual behavior alone (i.e., many pages allocated in a very short amount of time) could be sufficient to implement a solution to detect spraying attacks. For example, a simple approach could measure the speed of memory allocation and the average amount of memory usually allocated by the application under analysis. The first parameter would react to a quick memory increase, a common aspect of most of the existing attacks. The second parameter, once properly tuned for the application to protect, would act as a threshold of memory allocation, beyond which the behavior becomes suspicious and an alert is raised. It seems reasonable to believe that, by checking these two parameters, a detector could successfully prevent spraying attacks.

One of the main motivation of our work is to prove that the use of these two parameters is not sufficient for designing an effective spraying detection system. To prove our point, we designed a set of experiments to show that an attacker can tune the memory allocation behavior of an exploit to mimic the one of a normal application.

In our tests we used as a case study a classic heap spray attack against Internet Explorer 8 (described in CVE-2011-1996) but it is possible to replicate similar results with any applications where the memory allocation depends on input data. The first test we performed aimed at measuring the memory allocation curve while the user was visiting a small set of web sites. Figure 1 shows that the parallel execution of four common web applications (using parallel browser’s tabs) boosts the memory allocation of Internet Explorer to around 200MB. The same graph also shows the allocation curve of the CVE-2011-1996 exploit launched by Metasploit. In this case, the malicious behavior is easy to detect since it produces a huge allocation of memory in a short period of time – that then drops drastically after the success of the exploit. The drop is due to the fact that the shellcode spawns a process and releases the system resources of the previous execution thread along with its own memory. Other spraying attacks exhibit similar curves, a weakness that could be used to identify an ongoing malicious activity.

In the second experiment we wanted to answer two separate questions: i) how the total amount of memory allocated by the exploit affects the reliability of a spraying attack; and ii) whether it is possible for an attacker to slow down the attack in order to mimic the slope of the allocation curve observed on benign web pages. To

this end, we first modified our exploit to decrease the amount of sprayed memory. As we expected, reducing the number of allocated pages also reduces the probability of landing on one of them, thus making the exploit less reliable. We measured this phenomenon by running each exploit configuration ten consecutive times, counting in each case the number of successful attacks. The results of our tests show that the memory used by the original exploit can be largely reduced maintaining an acceptable success rate. For instance, the attack was still successful in 80% of the cases with a total memory consumption of only 131 MB – that is considerably less than what IE8 used in our benign scenario.

We then modified again the original exploit, this time introducing a delay between each memory allocation to mimic the behavior of a benign application. This change had no impact on the success rate of the attack. Figure 2 shows the allocation curve of our modified exploit, compared with a base line obtained by running Internet Explorer with six open tabs. From this experiment, it is clear that neither the speed nor the amount of memory can be used as the only criteria to detect a potential heap spraying exploit. By setting the threshold too low, the system would generate too many false alarms, and by raising the threshold too high the system would be vulnerable to evasions.

This conclusion motivates our further investigation to design a better memory monitoring and spraying attacks detection technique.

3 Preliminary Notions on Intel VT-x

Before we discuss our solution, we need to briefly introduce some virtualization concepts that we will use in the rest of the paper. Intel VT-x is a technology available in various Intel CPUs to support virtualization [23, 30].

VT-x defines two particular *transitions*: *vmexit*, to move from the guest to the hypervisor, and *vmentry*, to move in the opposite direction. As a result, the hypervisor is executed only when particular events in the guest trigger an exit transition. The set of events causing these transitions is extremely fine grained and can be configured by the hypervisor itself. Such events include exceptions, interrupts, I/O operations, and the execution of privileged instructions (e.g., accesses to control registers). Exits can also be *explicitly* requested by in-guest software, using the *vmcall* instruction. Because of its similarity to system calls, this approach is commonly called *hypercall*. Whenever an exit occurs, the hardware saves the state of the CPU in a data structure called Virtual Machine Control Structure (VMCS). The same structure also holds the set of exit-triggering events that

are currently enabled, as well as other control information of the hypervisor.

Another technology we need to introduce is the *Extended Page Tables* (EPT). This technology has been introduced to support memory virtualization, which is the main source of overhead when running a virtualized system. If enabled, the standard *virtual-to-physical* address translation is modified as follows. When a software in the guest references a virtual address, the address is translated into a physical address by the Memory Management Unit (MMU). However, the result of this operation is not a real physical address, but a *guest physical address* (*gpa*). The hardware then walks the EPT paging structures to translate the *gpa* into a *host physical address*, that corresponds to the actual physical address in the system memory. The EPT technology also defines two new exit transitions: *EPT Misconfiguration* and *EPT Violation*, respectively caused by wrong settings in EPT paging entries and by a guest attempting to access memory areas it is not allowed to. By altering the EPT entries, the hypervisor has full control of how the guest accesses physical memory. For example, it can remove write permissions from an entry, so that any write-access by the guest triggers a violation.

Threat Model

Our threat model considers an attacker that is able to exploit (either locally or remotely) an application running on the machine and to perform a spraying payload delivery. The use of a hypervisor-based technology is motivated by the goal of providing an OS-independent detection system and a more secure reference monitor.

Since we leverage late-launching to deploy our solution on operating systems running on physical machines, without requiring a reboot, we assume that the machine to be protected is clean when Graffiti is loaded. Thus, we consider the protection of already infected systems to be out of the scope of this paper.

4 Architecture Overview

In order to considerably improve over the state of the art, we set five main requirements for our detection system. First, it should be completely independent from the memory allocator used by the protected applications (R1). Second, it has to operate system-wide, i.e., it should be able to detect any memory allocation and deallocation that occurs in the system (R2) and it must be able to recognize any memory page that gets executed in the operating system (R3). Fourth, in order to operate correctly, our system should not require any OS-

dependent information (R4). Finally, the overhead introduced by the system should be reasonable, in line with other system-wide protection mechanisms. In particular, we consider a “reasonable” overhead, anything comparable to the one introduced by other virtualization systems such as XEN or VMware (R5).

To satisfy these five requirements, our system was designed to be easily extensible and configurable, and to tune its behavior (and therefore its overhead) to match the current level of risk of the monitored system. This is achieved by using two separate modes of operation.

Our monitoring platform is based on a custom hypervisor that normally runs in what we call *monitor mode*. In this mode, the hypervisor intercepts every new memory page that is allocated in the system, along with the CR3 register associated to the process that is requesting the memory. Whenever the total amount of memory requested by a single process exceeds a certain threshold (computed experimentally as described in Section 2.1) the system switches to *security mode* and starts performing additional checks to detect the presence of a possible attack for that particular application (while remaining in monitor mode for the other applications). In Section 2.1 we proved that a fixed threshold is not able to properly capture all the possible variations of spraying attacks. For this reason, in our system we use a threshold not for detection, but only to improve the performance of the system by disabling expensive checks when the total memory used by the process is too low for an attack to be successful. It is important to stress that in our solution, lowering the threshold for a given application does not introduce any false positives from the detection point of view, but only increases the overhead for that particular application alone (and not for the rest of the running system or for any other application).

When a process exceeds this minimum allocation threshold, the hypervisor performs two main tasks. First, using the EPT, it removes the execution permission from all the allocated pages, so that any attempt to execute code will be intercepted by the system. Second, it invokes the static analyzer component to check for the presence of a potential spraying attack. The actual detection is delegated to a configurable number of analysis plugins.

Figure 3 provides an overview of the system architecture and shows the interactions among the different components. The figure is divided in three parts, with user space on top, kernel space in the middle, and our custom hypervisor at the bottom. When an application (in this case a web browser) requests new memory, the kernel searches for a free page and it allocates it. At this point, when the OS tries to update the page table, the operation is intercepted by our hypervisor. If our system is

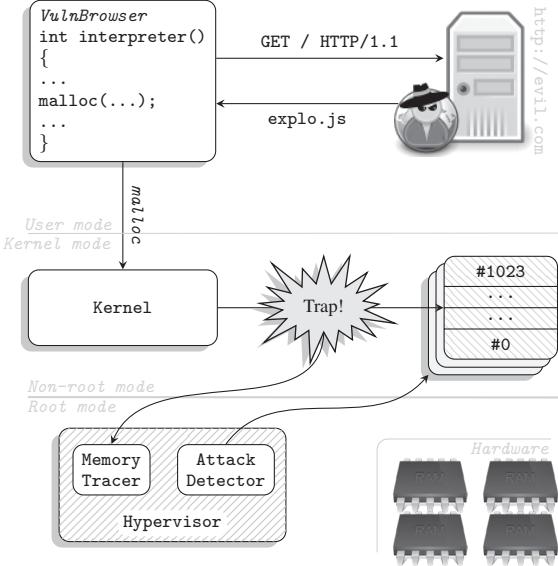


Figure 3: Architecture of the Memory Allocation Tracer.

running in monitor mode, the hypervisor only tracks the new memory allocation and gives back control to the operating system. If instead the application has already requested enough memory to trigger the security mode, our attack detection routines are executed to inspect the memory and flag any heap spraying attempt.

System Deployment

The main component that enables the protections enforced by Graffiti needs to keep an accurate track of all the allocation and deallocation operations that occur in the system. The main motivation of using a hypervisor is that, from a low level perspective, memory allocation is strictly dependent only on the hardware architecture, and not on the operating system itself. Thus, by working *below* the operating system, Graffiti avoids all the intricacies introduced by the various allocation engines, and therefore it does not require to modify or instrument the protected system (e.g., to place hooks inside OS components). Graffiti leverages late-launching to load its protection mechanism while the target is running. This hot-plug capability is achieved without rebooting the system, so it is transparent to the native OS. Finally, it is important to note that Graffiti is a very flexible system and can be configured according to the target needs. For instance, it can be deployed to monitor only a single sensitive process (e.g., a browser, or a PDF viewer), a set of thereof, or even the entire running system.

Our current prototype is implemented as an extension of HyperDbg, an open-source hardware-assisted hypervisor framework [17]. In Sections 5 and 6 we present

the design and implementation of the two main components of the system: the *Memory Tracer* and the *Attack Detection Routines*.

5 Memory Tracer

To implement our heap spraying protection technique, we must first keep track of all the allocation and deallocation operations that occur inside the system. Ideally, the most obvious solution to track memory allocations would be to modify the allocator itself, by extending the operating system with a new tracking feature. By doing so, however, our system would need to be customized for a particular operating system, and we would need to constantly update our tracker according to any OS upgrade.

To avoid this problem, we decided to implement our tracking approach at the hypervisor level (requirements R1 and R4), i.e., below the operating system. Since our approach is based on virtualization, from now on we will refer to the protected system alternatively with the term *guest* or *target*.

5.1 Tracer Design

Our system is designed to intercept every modification that is made by the guest OS to paging structures, and to recognize when the change corresponds to the creation or to the elimination of a page. To better illustrate our tracing technique, we will often refer to the paging structures that are used in the Intel architectures [23].

Whenever a process requires a new page, the kernel walks the paging structures of the requesting process looking for a usable Page Table Entry (PTE) in one of the Page Tables of the process (i.e., the second level structures). If none is found, it either allocates a new Page Table, by altering an entry on the first level paging structure (also known as the Page Directory), or it swaps some of the pages of the process to disk to create some empty slots. Once it has found or created a usable PTE, the kernel modifies it to map the allocated physical page to a virtual address, sets the lower 12 bits of the PTE to match the attributes of the page (e.g., `read/write`, `user/supervisor`), and returns the virtual address to the requesting process.

Our defense mechanism needs to keep a fine-grained view of every allocation to protect the system against spraying attacks. In particular, according to the address translation and new page allocation we need to intercept six different events: (1) Creation (2) Modification and (3) removal of a page. (4) Creation (5) Modification (6) Removal of a page table.

Whenever one of these six events is triggered by the kernel, our hypervisor intercepts the operation and acts accordingly. The first triple of events is traced to keep track of which pages a process allocates. The second group, on the other hand, must be traced to ensure that our system maintains a complete view of the allocated pages and does not miss any event in the first category.

5.2 Page Table Monitoring

Since Graffiti operates at the hypervisor level, it leverages the EPTs to write-protect all the page structures of a process. By doing so, it can intercept all modification attempts, as part of any of the six cases enumerated above. At first, the hypervisor detects when a new process is created by intercepting write operations to the CR3 register. As soon as a process is spawned by the kernel, a process will have just a limited number of paging structures, possibly inherited by its parent process (e.g., on Linux this depends on the flags of the `clone()` syscall that is used to spawn the process). To protect all its paging structures, Graffiti needs to traverse the page directory (pointed by the value of the CR3 register) and write-protect all the page tables pointed by each PDE. Page tables are scanned as well, to keep track of the physical pages allocated to the process by the kernel. After this setup phase is completed, each attempt to modify one of the pages would cause a trap in our hypervisor system.

Implementing the approach we just described while maintaining an acceptable overhead is a challenging task. At first, we use the EPTs to write-protect every paging structure of a target process. By doing so, whenever the OS kernel attempts to modify such structures because the process requires it (1), an EPT violation transfers the execution to the memory tracer component of our hypervisor framework (2). The violation is handled by removing the write protection and keeping a copy of the value of the entry (PTE or PDE) being modified (3), and re-executing the faulting instruction by performing an entry with the monitor trap flag (MTF [23]) raised (4). After the instruction has been executed, the hypervisor obtains again the control thanks to the exit caused by MTF (5), compares the new value stored in the entry with the old one and uses this information to infer which of the six kernel operations described previously has occurred (6). Eventually, the protection is restored (7) and the control is given back to the guest kernel (8).

To make the tracing mechanism clearer, consider the following scenario: the hypervisor intercepts a write attempt to the 2nd PTE of the 1st page table. This PTE originally contains the value 0. After single-stepping through the write instruction, we collect the new value of the PTE: `old:0x00000000 new:0xaffe007`.

This means that the guest kernel is mapping a physical page (at address 0xcaffe000) with a `rw` permission and making it accessible to both user and kernel space. In fact, the three lowest bits are set, making the entry present, writable, and accessible to user mode processes. For our framework, this operation corresponds to a *create page* event. To intercept when a process is created, we catch CR3 write operations in the guest. When the CR3 value that is going to be written corresponds to the one of a process we want to protect, we apply the protection to its paging structure, as explained above. It is important to stress that our approach is completely OS independent, as the only knowledge we rely on is the meaning of the bits stored in the paging structures, and those solely depend on the CPU architecture.

5.3 Graffiti Micro-Virtualization

The system described so far does not satisfy the requirement R2. In fact our solution should be able to monitor the entire system, and not only a few processes at a time. Unfortunately, by extending the previous approach to the whole guest operating system (all user-space processes and kernel threads), we observed a thrashing [4] phenomenon that introduced a large overhead in the memory allocation. This phenomenon creates a large number of context switches between OS and hypervisor, thus increasing the system overhead.

This phenomenon happens when a modification of a memory page of the running process creates as a side effect a modification of a memory page of another non-running process. This is a consequence of the fact that some memory pages are shared among processes, and some kernel tasks perform operations on memory pages of different processes. We refer to this problem as the *interference problem*.

The impact of this interference can be measured by running two simple tests. In the first, we computed the overhead introduced by our system while protecting a single process (Internet Explorer 10) and in the second we protected other two processes (Acrobat and Firefox) on top of Internet Explorer. The overhead on Internet Explorer alone went from 22% in the first test to 63% in the second, just as a side effect of monitoring two additional applications. Unfortunately, the interference of protecting more processes and the kernel itself would quickly slow down the entire system to a point in which it would not be usable anymore.

Ideally, we would like to design our system to avoid the interference problem, so that the overhead would not depend on the number of monitored processes. To achieve this goal, we propose a novel *micro-virtualization* technique, where each process runs inside

its own virtual memory sandbox and our tracking system enables the memory protection of just the process which is currently running. More in details, our micro-virtualization technique bases its approach on the fact that the VMCS contains a pointer to the EPT (EPTP) currently used by the hypervisor (see Section 3). Since we use the EPT to protect the processes (as explained in Section 5), our idea is to create a different EPT for each of the processes we protect, and change the EPTP in the VMCS at every context switch. From a low level perspective, this corresponds to intercepting every CR3 write operation (also easily trappable through VT [17]) and modifying the VMCS so that the EPTP points to the EPT of the process that has been scheduled for execution. Protected processes will have their own EPTs, while un-protected ones will just use a common EPT. To this end, every time a new process is created, the system creates a new EPT and associates it to the new process. It is important to note that the creation of this new EPT is not very costly, since the page table at the process creation is tiny and we only need to identify and protect some of them. By using such a mechanism, the hypervisor automatically disables the memory tracking of the other unprotected processes and enables the trapping only for the pages that are related to the currently protected processes, thus avoiding the thrashing side effect. Since this solution requires only to change the EPT pointer when a context switch occurs, it does not increase the overhead of the system.

In order to validate our micro-virtualization mechanism we performed two main experiments by using three applications: IE10, Acrobat Reader and Firefox. During our first experiment we only protect one application (IE10) and we compute the execution time and the overhead obtained by surfing several web pages in three main cases: (1) without hypervisor (2) with out hypervisor but without micro-virtualization and (3) with hypervisor and micro-virtualization enabled. From this first experiment the micro-virtualization does not introduce any additional overhead to the system when is used to protect a single process (23% in both cases with and without micro-virtualization). The only overhead introduced by the micro-virtualization occurs during the first loading of the new process. In this case the hypervisor needs to build up the EPT table for the new process by walking the process page tables. The overhead introduced during the loading time is 8%.

In the second experiment we test the scalability of our system with the new micro-virtualization mechanism enabled. This time we protect all three applications and we compute, like in the previous experiment, the execution time and the overhead obtained by surfing several web pages in the same three main cases: (1) with-

out monitoring the application (no hypervisor enabled) (2) with hypervisor but without micro-virtualization and (3) with hypervisor and micro-virtualization enabled. The overhead was 63% without micro-virtualization and 23% with micro-virtualization, confirming that the micro-virtualization is able to remove the overhead introduced by the *interference* problem.

As a result of our novel micro-virtualization architecture, our system is able to monitor an arbitrary number of different applications, without any increase in the system overhead. More specifically the overhead only applies to a particular protected application and it does not propagate to the rest of the system. For instance, if the user wants to protect only the browser and the PDF viewer against heap spraying attacks, any other application would not suffer any side effect or slowdown from our tracking system.

6 Detection Components

Whenever the total memory dynamically allocated by a process raises over a certain configurable threshold, the tracer switches to security mode and triggers a configurable number of static analysis routines to verify if a spraying attack is ongoing in the system.

Our current prototype includes three different components, presented in details in the next sections. These serve only as possible examples of the heuristics that can be easily plugged into our platform, and they could therefore be improved or extended with other techniques.

Malicious Code Detector

The aim of this component is to detect the simplest form of heap spraying. In this case, we assume the heap is randomized but executable, and therefore the attacker can spray the memory of the vulnerable target with multiple copies of a shellcode. Thus, the goal of this detector is to identify the presence of shellcodes inside the memory allocated by a process.

Our technique works as follows. First, the detector scans a fraction n of the most recently allocated memory pages and tries to disassemble them starting at twenty randomly selected offsets. For simplicity, any sequence of assembly instructions that terminates with a control transfer instruction that invokes a library call or system call is marked as a potential shellcode. To avoid cases where an attacker tries to obfuscate its attack by using an indirect control transfer instruction (iCTI), we consider each iCTI as a potential shellcode terminator.

The scan process is repeated for each allocated page and the detector finally reports the distribution of the

number of potential shellcode detected in each page. If the average number is higher than a given value, it raises an alarm. This approach derives from the observation that in the normal operation of a benign program only a small portion of the *analyzed* memory pages would contain a relevant fraction of valid instructions sequences. In an exploitation scenario, instead, most of the analyzed pages would contain close to 20 potential shellcode sequences. It is important to note that when the system starts disassembling from one page it continues till it reaches a code pointer, that may as well be located in a different page. If multiple pages are involved in such analysis they are all considered and marked as a shellcode container.

Self-unpacking Shellcode Detector

In this second scenario, we assume the same environment described before (ASLR enabled, DEP disabled), but we now consider the case in which an attacker packs her shellcode to make the detection more difficult. For example, all Metasploit payloads in spraying-assisted exploits are packed by default, e.g., by using the *shikata-ga-nai* encoder. Packed shellcodes are typically made up of a number of seemingly meaningless bytes, prepended with a small unpacking routine. The routine and the packed code are usually adjacent (i.e., they are located in the same memory page), as splitting them would lead to a waste of space and consequent loss of effectiveness when mounting the spraying attack.

Our second detection plugin is designed to detect packed shellcodes as soon as they start unpacking, and is tightly binded to the memory tracer. The component enforces what we call a *dynamic W \oplus X protection*. As soon as the memory tracer detects a new page allocation, it modifies the EPT entry corresponding to the newly allocated page so that a violation will be triggered when a write access to that page is attempted (R-X). The detector intercepts these attempts and modifies the EPT entry of the accessed page so that write accesses are enabled, but not execution accesses (RW-). If this new protection triggers a violation, we have a *write-then-execute* situation, which is fairly common in nowadays systems (especially with JIT engines). However, this mechanism allows to observe the more anomalous situation in which code modifies the same memory page in which it resides, that indicates the presence of self modifying code, used by packed shellcodes as described above. This technique is also effective when DEP is enabled on the heap memory, and the attacker uses a JIT-spraying attack. In fact, if the JIT-sprayed payload is packed, it will need to unpack itself and thus will trigger our detection heuristic.

Data Spraying Detector

When DEP is enabled, and JIT spraying is not a viable solution (e.g., there is no JIT engine in the vulnerable process), a possible exploit solution is to use return oriented programming. In this case, the attacker no longer sprays the heap with executable code but instead with multiple copies of a ROP chain. To trigger the code, the attacker then uses a *pivoting sequence* to move the stack pointer into the heap and let execution slide down the ROP chain, as we explained in Section 2.

To detect data spraying attacks, we designed a component that samples the most recently allocated memory pages of a process, and it considers any word inside them as a potential memory address. For each of these candidate addresses, the data spraying detector checks whether this address points to a valid executable page, and, if so, marks it as a *potential code pointer*. In case the total number of code pointers for each page is over a threshold, the system raises an alarm.

Unfortunately, even though this policy may sound reasonable at a first glance, we observed that in practice it suffers from a large amount of both false positives and false negatives. The first problem is related to the fact that modern operating systems use different techniques to load pages, one of which is called *Demand Paging* [4]. In this case the pages are only brought into memory when the running-process demands them. This optimization creates an issue for our detection method because when the system extracts the potential code pointers from the memory pages and checks if they point to a valid code page, the page may not be present in the page table (even if it is properly allocated). We observed this behavior during our experiments, and the result is that certain addresses would be discarded—thus potentially creating *false negatives* by missing a page that is part of a spraying attack.

To avoid this issue, we modified our hypervisor to intercept page faults in the guest system when Graffiti switches to security mode for a given process. When the detector checks an address that points to a memory page that is not mapped, the system does not discard it but keeps it as a potential code pointer into the memory structures of the hypervisor. Afterwards, when the process gets access to the demanded page, the system loads it and our detection system intercepts the page faults and it checks if the potential code pointer points to this memory page. If true, the system marks all the memory pages previously allocated that contain such address as suspicious and then it re-applies again the previous technique on the new set of pages.

The second problem of our original technique is the high number of false positive we observed in the experiments because benign memory pages also contain a sig-

nificant number of code pointers (e.g., in case of C++ classes or arrays). To reduce the false positives created by those benign memory pages, we improved our detector algorithm by replacing the pointers counter with a more sophisticated pointers frequency analysis. The idea is to compute the frequency of the code pointers that *every* page of the entire set contains, instead of analyzing every page individually. While the absolute number of code pointers may be deceiving, we observed that the distribution of those pointers in case of benign applications is really diverse, while in case of an attack the distribution tends to be quite uniform.

7 Experimental Results

The goal of our experiments is to first measure the overhead of the system in a realistic environment and then to show how effective our heuristics are in distinguishing spraying attacks from a normal allocation behavior.

Our code is composed by three main software components: a core hypervisor framework based on HyperDBG, the micro-virtualization implementation, and the detector plugins. The core hypervisor framework is written in a combination of C (17353 LoC) and assembly (545 LoC). The micro-virtualization and detector components account for 1435 lines of C programming language.

All tests presented in this section were performed on two machines, equipped with an Intel Core i5-2500 @ 3.3 GHz and 8GB of RAM, running respectively Windows 7 Professional 32bit and Debian Wheezy 32bit (kernel 3.2).

Activation Threshold and Overhead

Our system is designed to be adaptive. Consequently, the only part that is always active is the Memory Tracer. Our micro-virtualization solution confines the overhead to a single process and allows our system to monitor an arbitrary number of different applications without any increase in the overhead of the rest of the system.

During normal operation, the tracker overhead is negligible, and it is only noticeable when the monitored application allocates tens of megabytes of memory at a time – typically at start up or when a large document is open. To measure this worst case scenario, we used the stress suite to simulate a program that intensively allocates memory on a Windows 7 and on a Linux 3.2 hosts at a rate of 8MB every 2 seconds. The overhead we observed during the allocation phase was of 24% on Windows and 25% on Linux for a single process without considering context switch. Again, it is important to

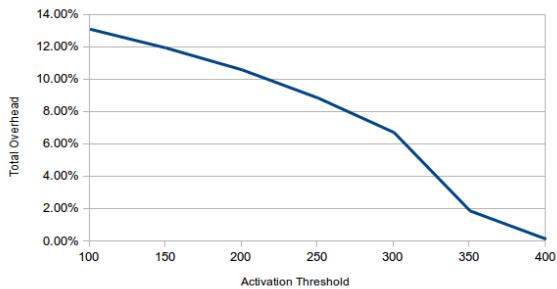


Figure 4: Detection Overhead for Internet Explorer 8

note the experiments performed in these tests produced a very intensive memory allocation activity and it is not representative of the memory behavior of the entire life of a process.

On top of this overhead, each application can observe a different overhead when Graffiti switches to security mode and enables the detection modules to scan the application memory. The frequency at which this happens depends on the value of the activation threshold. The lower the threshold, the hardest it is for an attacker to evade detection – but the higher the *potential* overhead for the application. “Potential” in this context means that the actual overhead also depends on the application: some use so little memory during their normal operation that the security mode would never be triggered - also for very low values of the threshold. Moreover, most of the applications only allocate large amount of memory when the user opens a document, but then the use of memory becomes quite constant - and therefore Graffiti’s negative impact tends to be concentrated only on the few initial seconds, and becomes negligible after that.

To measure this trade-off we performed two experiments. In the first, we asked some users to surf the web by using Internet Explorer 8 on Windows 7 with our detection system activated. We choose IE8 since this application usually uses a large amount of memory and it represents one of the main targets of spraying attacks. To mimic a realistic behavior, the users kept a tab open on GMail, and then alternately opened three other tabs performing memory intensive activities: watching videos on YouTube, browsing Facebook, and checking hundreds of pictures on 9gag. In the second experiment, we used Acrobat Reader for Linux to open 100 benign PDF files including conference papers, books, Ph.D. dissertations, and very large manuals (i.e., the Intel Manuals).

Following the approach used by Nozzle [36], we selected a sampling rate of 10% (number of pages checked

by our detection module over the total number of pages allocated). As a reference, with this value Nozzle introduced an overhead of 20% to Internet Explorer. The overhead obtained with our system is shown in Figure 4 for different values of threshold. Also in the worst case with the activation threshold set to 150MB, the overhead was only 12%. Moreover, the heuristic responsible for most of this overhead is the one that requires to randomly disassemble the content of the memory pages. Since this component is useless on any modern OS when DEP is enabled, the detection overhead of Graffiti becomes barely noticeable.

Moreover, our experiments with Acrobat Reader never reached the activation threshold, even when it was set at the conservative value of 100MB. In this case, the overhead of Graffiti on the normal use of the application was constantly zero – showing that for some popular applications our framework can provide a very complete protection against known and unknown attacks with no additional overhead.

Detection Accuracy

To test the effectiveness of our system, we measured the true and false positives rates for each individual detection technique that is currently included in the Graffiti prototype. To test the detection rate we used several real world exploits that cover all the different spraying techniques and variations mentioned in this paper. It is important to note that the six attacks that we chose for our experiments, summarized in Table 3, are representative for the entire spectrum of the techniques used by the spraying attacks described in Section 2. On top of this qualitative test, we also performed a quantitative test using over 1000 different malicious PDF documents that rely on heap spraying in the exploitation phase.

In the first test we show the effectiveness of our system to detect exploits based on stack pivoting, by using the attack described in CVE-2011-1996. The attack first sprays the stack frames on the heap and then executes a number of ROP gadgets in order to disable the DEP protection. During the spraying phase the attack allocates on average 384MB.

In this case, the static analyzer applied the code pointers frequency analysis on the attack memory pages. The component detected a high number of code pointers with a variance close to 0 in all the allocated memory pages, and thus it raised an alert successfully preventing the attack. To evaluate the false positive of such technique, we instructed our detector to track all the memory pages allocated by Internet Explorer 8 while browsing the first 1000 top Alexa domains [1]. In this case, the frequency of code pointers had a very high variance on all memory pages captured by the system, thus generating zero

Web Domain	Average	Variance
amazon.com	3	259.30
ask.com	7	867.90
baidu.com	8	559.57
blogspot.com	2	158.88
craigslist.org	6	391.15
delta-search.com	8	809.21
facebook.com	23	3521.68
google.co.jp	10	562.99
google.com.br	7	459.57
google.com	10	46.44
instagram.com	14	2763.22
microsoft.com	5	395.72
msn.com	16	2916.28
yahoo.com	14	1183.43

Table 1: Code Pointer Frequency Analysis Results.

Web Domain	Shellcode per page	Average Guess Offset
amazon.com	10/500	1/20
ask.com	2/500	1/20
baidu.com	48/500	4/20
blogspot.com	64/500	4/20
craigslist.org	10/500	2/20
delta-search.com	8/500	3/20
facebook.com	25/500	4/20
google.co.jp	162/500	3/20
google.com.br	69/500	3/20
google.com	74/500	3/20
instagram.com	224/500	5/20
microsoft.com	0/500	—
msn.com	5/500	1/20
yahoo.com	13/500	1/20

Table 2: Shellcode Frequency Analysis Results.

false alarms (Table 1 reports the results for the first 14 domains analyzed).

In the second set of experiments we tested the effectiveness of the Malicious Code detection component. In this case we used the exploit for CVE-2009-2477 affecting the Javascript interpreter of Mozilla Firefox 3.5. This attack exploits a memory corruption vulnerability in the Firefox browser, in which the Javascript interpreter fails to preserve the return value of the `escape()` function and results in the use of an uninitialized memory area. During the exploit, Graffiti reported that the average number of analyzed pages containing a potential shellcode was 100% – thus raising an alarm and stopping the attack. To test the false positive of the same detection technique, we used the same browser to visit the top 1000 Alexa domains. In this case, the average number of potential shellcodes per page was always below 50% and therefore no false alert were raised in the test. In table 2 we reported results about the first 14 domains analyzed, the number of pages that present potential shellcode and the average on shellcode found in the first 500 allocated memory pages. As we can see from the table, our malicious code detector component does not present any false positive.

CVE	Application	Exploit Technique	Detected
2010-0248	Adobe Flash player	ROP + packed sc	Yes
2011-0609	Adobe Reader	JIT + packed sc	Yes
2011-2462	Adobe Reader	ROP + packed sc	Yes
2010-2883	Adobe Reader	Ret2Lib + packed sc	Yes
2011-1996	IEexplorer	ROP	Yes
2009-2477	Firefox	Plain Shellcode	Yes

Table 3: Exploitation Detection Results.

In our third experiment, we analyzed the Self-unpacking Shellcode Detector component. In this case we selected different CVEs and we used the metasploit [35] tool to exploit them with packed payloads. In particular, we used two packing methods: the shikata-ga-nai packer and a simple xor algorithm. Our detection system was always able to intercept the first execution of the packed code and consequently detect the attacks without any false negative. Also for this component, we tested the false positive rate by browsing the top 1000 domains from the Alexa dataset. We did not observe any false positive, even though several website included obfuscated Javascript code. A further investigation on obfuscated java-script shows that the de-obfuscation routine is implemented at the compiler level so it does not present any problem or generate any false positive in our system.

To conclude, Table 3 reports all the vulnerabilities we used for our tests, along with the type of payload delivery and the detection results of Graffiti. Even though our detectors had a very high precision in all our tests, an attacker equipped with knowledge about the internals of our detectors could try to mimic the behavior of a benign application to evade detection. A further analysis of such attacks is presented in Section 8.

Aggregated Experiments

So far, we tested each piece of our infrastructure in isolation. In our final experiments, we put all pieces together. In the first test, we used Graffiti to analyze three datasets: a set of 1000 malicious PDF documents, a set containing 1000 benign web pages, and one containing 1000 benign PDFs. The first dataset was collected by a company working on malware analysis, while the other included the top Alexa web pages and random documents collected from various sources. All experiments were conducted with a very conservative activation threshold of 150MB and a sampling rate of 10%. Graffiti successfully detected all malicious documents, with zero false alarms. Moreover, the overhead on loading the web pages was in average of 23% (a value in line with previous OS-specific approaches that were only able to protect the web browser).

In the second experiment, we asked real users to use a Graffiti-protected system during their everyday activities for a total of 8-to-10 hours per day in a 7-days period. Graffiti was installed on two Windows 7 machines, configured to monitor Internet Explorer 8 with an activation threshold of 150 MB. All three spraying attacks detectors were enabled during the experiments (even though the first was not necessary on this setup). Overall, the real users visited a total of 492 distinct web pages and the detectors were activated 55 times, with an average of ≈ 8 times per day. On the same period, Graffiti raised 12 alerts on pages that seemed to be benign. A closer inspection of the FPs showed the data spraying detector (Section 6) to be the only responsible. This component is in charge of detecting data spraying attacks and bases its detection on the number of potential code pointers present in memory.

It is important to stress the fact that the three detection plugins are not the main contribution of our work, and our micro-virtualization framework allows other researchers to easily improve, extend, and replace them with other techniques. For instance, a possibility to decrease the false positive rate of this component could be to check not only if the code pointer points to a correct executable page, but also whether it points to a dangerous machine instruction sequences (e.g., a gadget). We manually inspected the websites that raised the false alarms and we found that applying such simple method would be able to prevent all the alerts. This check could be activated only when the data spraying detector identifies a possible attack, to prevent a significant increase of the overhead.

8 Security Evaluation

It is possible that an attacker, knowing the internals of our three detectors, could mount a mimicry attack that can successfully evade detection. For instance, an attacker can elude the code pointers frequency analysis by mimicking the variance of benign memory pages. Although this technique can be successful, it has two restrictions. The first is related to the minimum number of gadgets that the attacker needs to connect to perform a useful attack. To be useful, an attack should execute either an API call or a system call. Based on the number of API call parameters, we estimate that a useful number of gadgets for a standard shellcode is around 20 (i.e., to call the `VirtualProtect` function, commonly used in Windows shellcodes to remap a page as executable) even though some previous works show that the length of the gadgets for useful shellcode may vary from 8 to 12 [32]. The second restriction is related to the maximum number of gadgets that an attacker can include to

build a shellcode. Theoretically this number could be infinite. In case of spraying attacks, to increase the chance of success, the size of the shellcode should be smaller than the size of the NOP-sled, otherwise the probability to divert the control-flow of the application to the appropriate entry points decreases. In our experiments, for benign applications the range of code pointers in memory varies between 0 and 1024, with the vast majority of pages on the left end of the scale. These values are hard to mimic in a real attack. It may be possible in certain particular cases, but still our component would have considerably raised the bar making the exploitation much more difficult.

Another way an attacker can avoid detection is by evading the shellcode frequency analysis. To this end, the attacker can act on two parameters. She can decrease the number of successful entry points for each page – but this would drastically decrease the success rate of the attack. A second, more subtle, technique would consist in spraying the memory only with NOP instructions, and inject only one copy of the shellcode in a second time, when Graffiti already concluded its analysis. In this case we could extend our component to postpone the analysis when long nop sequences are identified. It is important to note that the attacker cannot wait for a long time in order to inject the shellcode, since any additional memory allocation done by the application would break the continuity of the nop sled.

A current limitations of Graffiti is that it cannot handle the case when an application allocates a big chunk of memory at the beginning of the process and then uses its own allocation functions to perform memory operations. However, since none of the applications that we tested in our experiments exhibited such behavior, we left this case for a future improvement.

To summarize, we believe that evading our three heuristics is not easy but it is certainly possible. However, the contribution of this paper is not in the heuristics per se, but in the underlying monitoring framework. Graffiti offers the first comprehensive, multi-OS solution, and this is an important step forward compared with existing defense solutions and compared with other techniques presented in previous papers.

9 Related Work

Several solutions have been proposed so far to cope with single instances of the spraying problem. In the following we summarize the existing works that address heap, JIT, and data spraying techniques.

Heap Spraying

Researchers have proposed several approaches for detecting heap-spraying attacks [36, 16, 21]. For example, Egele et al. [16] used x86 emulation techniques to defend web browsers against drive-by download attacks that use heap-spraying code injection. More in details, the authors proposed to check for the presence of a shellcode by monitoring all the strings that are allocated by the JavaScript interpreter. Their goal is similar to that of NOZZLE [36], which uses static analysis of the objects on the heap to detect heap-spraying attacks. In particular, NOZZLE scans memory objects looking for a sequence of instructions that includes a NOP sled and ends with a malicious shellcode. However, as the authors point out, the tool presents several drawbacks. For example, attackers can evade detection by avoid using large NOP sleds. Moreover, NOZZLE is also specific for the Java Script Engine Memory Allocator and it cannot be applied to a generic application. Another work to defend against heap spraying attack is BuBBLE [21]. In this case, the authors start from the assumption that an attack needs to spray a large part of the Heap memory with homogeneous data (i.e. NOP sled). BuBBLE breaks such an assumption by inserting special values in a random position inside strings before storing them in memory, and removing them when a string is used by the application. Again this solution is specific for the Javascript language and it cannot be easily ported for the protection of other applications.

Our approach is different since it does not require to know how the memory allocator of a particular interpreter engine works, and consequently it does not require access to source code and it is operating system independent. Moreover, it can protect any system application as well as kernel subsystems without any assumption about internals of the protected component.

JIT Spraying

Bania [5] proposed a detection technique based on the fact that in order to force the JIT compiler to generate code, an attacker should use ActionScript arithmetic operators. However, it is not mandatory for JIT spraying attacks to use arithmetic operations.

Another JIT spraying defense has been proposed by Hu et al. [22]. This solution consists of a kernel patch, JITsec, that tests for several conditions when a system call is invoked. In particular, the authors argue that an application can maintain its security properties and execute code from the stack and heap by decoupling sensitive from non-sensitive code and allowing the latter to run from writable memory pages. As a result, such detector only detects attacks that directly issue system

calls. Mimicry attack and ROP attacks are therefore not covered by this model.

JITDefender [11] is another work based on hardware-assisted technologies which aims at defeating JIT Spraying attacks. The system protects the Virtual Machine dynamic memory pages created by the JIT-Compiler and allows to execute only the pages requested by the VM. This approach is strictly VM dependent, and it can only detect JIT-spraying attack.

Our solution is orthogonal to the type of attack, and therefore it can successfully detect JIT-spraying attacks without any assumption about the instructions that are used by the attacker.

Finally, Lobotomy [27] proposes to mitigate JIT spraying attacks by applying the principle of least-privilege to the Firefox JIT engine: by splitting the compiler and executor modules of the engine, indeed, it greatly reduces the amount of code that needs to access writable *and* executable pages. The main drawbacks of Lobotomy, with respect to Graffiti, are: 1) its overhead, that is sensibly high if compared with ours, and 2) the need to re-design the JIT engine of the protected process. The latter is particularly hindering because it greatly limits the portability of Lobotomy to other JIT engines. On the contrary, Graffiti can seamlessly protect *any program*, without modifying any of its inner components.

Data Spraying

Several defensive solutions have been proposed to avoid pivoting-based techniques [28, 33, 34]. One of the most deployed is part of EMET [28], a solution designed by Microsoft. EMET is a utility that helps to prevent vulnerabilities in software from being successfully exploited. Among other features, EMET also addresses the problem of stack pivoting attacks by checking if the stack pointer points outside of a process stack boundaries whenever a dangerous API is invoked. However, several researchers proved that it is possible to bypass the EMET technology in many ways [24, 18, 37]. The impact of these studies show that technologies that operate at the same level of execution of the malicious code need to be extensively tested and carefully designed to offer the desired protection and avoid possible bypasses. Consequently, these studies also shows the importance of designing reference monitors that operate at a lower level (e.g., at the hypervisor level) such as Graffiti to avoid these trivial attacks.

Moreover, Microsoft recently introduced two new countermeasures to hinder browser exploitation: isolated heap and delayed free [25, 45]. Both these techniques raise the bar for use-after-free attacks; as stated by the Fortinet Labs researchers [19], they also make

heap manipulation harder, but they are not a general solution as they protect only the Internet Explorer browser.

10 Conclusion

In this paper we propose an efficient and comprehensive solution to defeat spraying attacks by tracking the memory allocations of the system in an OS-independent way.

Overall, our paper makes several contributions: we introduce the concept of micro-virtualization that allows us to design an efficient and effective memory allocator tracker. We presented Graffiti, a general and extensible memory analysis framework that has good performance and it is freely available and open source. On top of it, we created three heuristics to detect and prevent spraying attacks. However, we believe that in the future Graffiti can also be extended and adopted in other domains, such as malware analysis or memory forensics.

References

- [1] Alexa top domains. <http://www.alexa.com/topsites/category/>.
- [2] Rop attack against data execution prevention technology, 2009. <http://www.h-online.com/security/news/item/Exploit-s-new-technology-trick-dodges-memory-protection-959253.html>.
- [3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.
- [4] Greg Gagne Avi Silberschatz, Peter Baer Galvin. Operating system concepts. <http://os-book.com/>.
- [5] Piotr Bania. Jit spraying and mitigations. *arXiv preprint arXiv:1009.1038*, 2010.
- [6] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. *SIGPLAN Not.*, 41(6):158–168, June 2006.
- [7] Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.
- [8] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.
- [9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy*, pages 380–394, May 2012.
- [10] Liang Chen and Qidan He. Shooting the osx el capitan kernel like a sniper, 2016. <https://speakerdeck.com/flankerhqd/shooting-the-osx-el-capitan-kernel-like-a-sniper>.
- [11] Ping Chen, Yi Fang, Bing Mao, and Li Xie. Jitdefender: A defense against jit spraying attacks. In Jan Camenisch, Simone Fischer-Hbner, Yuko Murrayama, Armand Portmann, and Carlos Rieder, editors, *SEC*, volume 354 of *IFIP Advances in Information and Communication Technology*. Springer, 2011.
- [12] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-hartman, Mike Frantzen, and Jamie Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *In Proceedings of the 10th USENIX Security Symposium*, 2001.
- [13] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *In Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.
- [14] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with javascript, 2008.
- [15] eEye Research. Microsoft internet information services remote buffer overflow (system level access), 2001. <https://web.archive.org/web/20061026101830/http://research.eeye.com/html/advisories/published/AD20010618.html>.
- [16] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of*

- the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA '09*, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag.
- [17] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, Antwerp, Belgium, September 2010. <https://code.google.com/p/hyperdbg/>.
 - [18] Fireeye. Using emet to disable emet. https://www.fireeye.com/blog/threat-research/2016/02/using_emet_to_disable.html.
 - [19] Fortinet Labs. Is use-after-free exploitation dead? The new IE memory protector will tell you. <http://blog.fortinet.com/>.
 - [20] Ivan Fratic. Exploiting internet explorer 11 64-bit on windows 8.1 preview, 2013. <https://ifsec.blogspot.com/2013/11/exploiting-internet-explorer-11-64-bit.html>.
 - [21] Francesco Gadaleta, Yves Younan, and Wouter Joosen. Bubble: a Javascript engine level counter-measure against heap-spraying attacks. In Fabio Massacci, Dan Wallach, and Nicola Zannone, editors, *ESSoS, Pisa, 3-4 February 2010*. Springer Berlin / Heidelberg, January 2010.
 - [22] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments, VEE '06*, pages 2–12, New York, NY, USA, 2006. ACM.
 - [23] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3 (3A,3B,3C combined)*, March 2013.
 - [24] Bromium Labs. Bypassing emet 4.1. <http://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf>.
 - [25] MWR Labs. Isolated heap & friends - object allocation hardening in web browsers. <https://labs.mwrinfosecurity.com/blog/2014/06/20/isolated-heap-friends---object-allocation-hardening-in-web-browsers/>.
 - [26] Lixin Li, James E. Just, and R. Sekar. Address-space randomization for windows systems. In *ACSAC*, pages 329–338. IEEE Computer Society, 2006.
 - [27] Jauernig Martin, Neugschwandtner Matthias, Milani-Comparetti Paolo, and Christian Platzer. Lobotomy: An Architecture for JIT Spraying Mitigation. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, September 2014.
 - [28] Microsoft. The enhanced mitigation experience toolkit. <http://support.microsoft.com/kb/2458544>.
 - [29] Microsoft. Structured exception handling overwrite protection (sehop). <http://support.microsoft.com/kb/956607>.
 - [30] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.
 - [31] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: automatically correcting memory errors with high probability. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 1–11. ACM, 2007.
 - [32] Michalis Polychronakis and Angelos D Keromytis. Rop payload detection using speculative code execution. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 58–65. IEEE, 2011.
 - [33] Aravind Prakash and Heng Yin. Defeating rop through denial of stack pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference, ACSAC 2015*, pages 111–120, New York, NY, USA, 2015. ACM.
 - [34] Rui Qiao, Mingwei Zhang, and R. Sekar. A principled approach for rop defense. In *Proceedings of the 31st Annual Computer Security Applications Conference, ACSAC 2015*, pages 101–110, New York, NY, USA, 2015. ACM.
 - [35] Rapid 7. Metasploit penetration testing software. <http://www.metasploit.com>.
 - [36] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the Usenix Security Symposium*, August 2009.

- [37] Duo Security. Wow64 and so can you bypassing emet with a single instruction. <https://duo.com/assets/pdf/wow-64-and-so-can-you.pdf>.
- [38] Skylined. Microsoft internet explorer 6 - (iframe tag) buffer overflow exploit, 2004. <https://www.exploit-db.com/exploits/612/>.
- [39] Skylined. Heap spraying high addresses in 32-bit chrome/firefox on 64-bit windows, 2016. <http://blog.skylined.nl/20160622001.html>.
- [40] Kevin Snow, Srinivas Krishnan, Fabian Monroe, and Niels Provos. Shello: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, 2011.
- [41] Kevin Z. Snow, Fabian Monroe, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.
- [42] Alexander Sotirov. Heap feng shui in javascript, 2007.
- [43] The PaX Team. Pax address space layout randomization. Technical report <http://pax.grsecurity.net/docs/aslr.txt>.
- [44] Team Teso. 7350854.c, 2001. <https://www.exploit-db.com/exploits/409/>.
- [45] Trendmicro Labs. Mitigating UAF Exploits with Delay Free for Internet Explorer. <http://blog.trendmicro.com/trendlabs-security-intelligence/mitigating-uaf-exploits-with-delay-free-for-internet-explorer/>.
- [46] Vupen. Microsoft Internet Explorer javaprxy.dll COM Object Vulnerability / Exploit (Security Advisories). <http://www.vupen.com/english/advisories/2005/0935>.
- [47] Vupen. Microsoft Internet Explorer "Msdds.dll" Remote Code Execution / Exploit (Security Advisories). <http://www.vupen.com/english/advisories/2005/1450>.

Request and Conquer: Exposing Cross-Origin Resource Size

Tom Van Goethem, Mathy Vanhoef, Frank Piessens, Wouter Joosen
iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
first.lastname@cs.kuleuven.be

Abstract

Numerous initiatives are encouraging website owners to enable and enforce TLS encryption for the communication between the server and their users. Although this encryption, when configured properly, completely prevents adversaries from disclosing the content of the traffic, certain features are not concealed, most notably the size of messages. As modern-day web applications tend to provide users with a view that is tailored to the information they entrust these web services with, it is clear that knowing the size of specific resources, an adversary can easily uncover personal and sensitive information.

In this paper, we explore various techniques that can be employed to reveal the size of resources. As a result of this in-depth analysis, we discover several design flaws in the storage mechanisms of browsers, which allows an adversary to expose the exact size of any resource in mere seconds. Furthermore, we report on a novel size-exposing technique against Wi-Fi networks. We evaluate the severity of our attacks, and show their worrying consequences in multiple real-world attack scenarios. Furthermore, we propose an improved design for browser storage, and explore other viable solutions that can thwart size-exposing attacks.

1 Introduction

In 1996, Wagner and Schneier performed an analysis of the SSL 3.0 protocol [67]. In their research, the authors make the observation that although the content is encrypted, an observer can still obtain the size of the requested URL as well as the corresponding response size. The researchers further elaborate that because it is possible to make an inventory of all publicly available data on a website, knowing the size of requests and responses allows an attacker to determine which web page was visited. Although content is increasingly being served over secure SSL/TLS channels [55], the length of requests

and responses remains visible to a man-in-the-middle attacker. Consequently, the attack that was described by Wagner and Schneier two decades ago remains universally applicable. However, due to the various transitions the web underwent, the consequences of uncovering the size of remote resources have shifted drastically.

With the advent of online social networks, the dynamic generation of web pages goes even further. When browsing, each user is now presented with a personalized version, tailored to their personal preferences and information they, or members of their online environment, (un)willingly shared with these online services. Consequently, the resource that is returned when a user requests a certain URL will often reflect the state of that user.

Two types of size-exposing attacks, namely traffic analysis and timing attacks, have been widely studied. In traffic analysis, an adversary passively observes the network traffic that is generated by the victim's browsing behavior. Based on the observed size, sequence, and timing of requests and responses, an attacker can learn which website was visited by the victim [13, 25, 68], or uncover which search queries the user entered [12, 40]. In contrast to traffic analysis, where the threat model is typically defined as a passive network observer, launching a web-based timing attack requires the adversary to trick the victim in making requests to certain endpoints, which is typically achieved by running JavaScript code in the victim's browser. The attacker then measures the time needed for the victim to download the specified resources, which, depending on the victim's network condition, allows him to approximate the resource size, and ultimately obtain information on the state of the user [19, 7, 14].

Motivated by the severe consequences on the online privacy of a vast amount of users, we present a systematic analysis of possible attack vectors that allow an adversary to uncover the size of a resource. As a result of this evaluation, we discover design flaws in various browser features that allow an adversary to uncover the exact size

of any resource. Furthermore, we demonstrate that by intercepting and manipulating encrypted Wi-Fi traffic, an adversary can uncover the exact size of an HTTP response. By leveraging these techniques, we show that when an attacker can make the victim send requests to arbitrarily chosen endpoints, the potential consequences of traffic monitoring become significantly more severe. In contrast to prior attacks, where adversaries could typically only obtain a rough estimate of the resource size, or were unable to attribute network traffic to specific requests, our size-exposing attacks show that the capabilities of an adversary are worryingly extensive, as we exemplify by the means of several real-world attack scenarios. Finally, we explore the viability of several defense mechanisms, leading to an improved browser design and a variety of possibilities for websites to thwart size-exposing attacks.

Our main contributions are:

- We perform an in-depth analysis at the level of the browser, network and operating system, and explore techniques that can expose the size of resources either directly or through a side-channel attack.
- We introduce several new attack vectors that can be leveraged to uncover the exact response size of arbitrarily chosen endpoints.
- By the means of several attack scenarios on high-profile websites, we demonstrate that an adversary can reveal the unique identity of an unwitting visitor within mere seconds, and extract sensitive information that the user shared with a trusted website.
- We propose an improvement to the specification of the Storage API, and explore various existing solutions that can be used to mitigate all variations of size-exposing attacks.

The remainder of the paper is structured as follows: in Section 2 we provide a high-level overview of the technical aspects related to recently introduced browser features. In Section 3 we present an in-depth analysis on potential size-exposing techniques, and elaborate on how these can be used in various attack scenarios. In Section 4, we discuss how adversaries can leverage these techniques against a number of real-world services. Furthermore, in Section 5 we propose and explore methods that can thwart size-exposing attacks. Section 6 covers related work, and Section 7 concludes this paper.

2 Background

One of the most important security concepts of modern browsers, is the notion of Same-Origin Policy [64, 73].

Despite what its name may suggest, it is not strictly defined as a policy, but rather represented as certain principles that ensure websites are restricted in the way they can interact with resources from a different origin. Although it is possible to initiate a cross-origin request, the Same-Origin Policy prevents reading out the content of the associated response, which is obviously imperative in order to provide online security. Naturally, the content of resources is not the only part that should be shielded off from other origins; the size of a resource should also be considered sensitive, as evidenced by the several case studies presented in Section 4 and prior work [7, 20, 60]. As such, it comes as no surprise that the browser APIs that are responsible for making HTTP requests will only report the length of a response when the associated request was to the same origin.

The Fetch API, which is currently implemented by Google Chrome, Firefox and Opera, and is under development by other browser vendors [39, 69], introduces a set of new semantics that aim to unify the fetching process in browsers. In short, the `fetch()` method is given a `Request` object, and a second, optional parameter that specifies additional options for the request. For instance, when the `credentials` option is set to "`include`", the user's cookies will be sent along with the request, even when it is cross-origin. The `fetch()` operation will return a `Promise` that yields a `Response` object as soon as the response has been fetched. In case the request was authenticated, cross-origin, and did not use the CORS mechanism, i.e., the `mode` was set to its default value "`no-cors`", the `Response` will be marked as "`opaque`", which will mask all information (status code, response headers, cache state, body and length) of the response to prevent cross-origin information leakage. In the following sections we will show how certain browser mechanisms can be abused to uncover the length of cross-origin responses.

Although the content of an opaque `Response` can not be accessed, it is possible to force the browser to cache that resource. The Cache API, which is part of the Service Worker API [66], can be used to place `Response` objects in the browser's cache. For security purposes, the cache that is accessed by the Cache API is completely isolated from the browser's HTTP cache, and is not shared across different origins. The cache is accessed by opening a `Cache` object, which can then be used to store `Response` objects with their associated `Requests`. Note that *any* response can be stored, regardless of the `Cache-Control` headers sent out by the web server. To prevent a malicious entity from completely filling up the user's hard disk, certain quota rules apply. The details of these rules will be explained in more detail in Section 3.4.

3 Size-exposing Techniques

As demonstrated by prior research, the size of a website’s resources is often related to the state of the user at that website [7, 38, 60]. Consequently, knowing the size of these resources allows an adversary to (partially) uncover the state of the user, which often yields sensitive information. In order to detect the presence of size-exposing attack vectors, we performed an in-depth analysis on all operations in which resources are involved. In this section, we present the results of this analysis and discuss the various techniques that can be used to infer the size of cross-origin resources. Next to the size-exposing methods that were discovered in prior research, we also introduce various novel techniques and re-evaluate methods in the light of recent protocol evolutions.

Throughout this section, we consider different attacker models based on the evaluated resource operation. As a rule of thumb, for each resource operation we considered all the attacker models in which the adversary is able to make observations about the operation. For instance, when analysing the transfer of a resource over the network, multiple attacker models were taken into account: an eavesdropper might inspect the encrypted network traffic directly, or Wi-Fi packets could be examined when the adversary is in physical proximity of the victim, or the attacker might simply use JavaScript to measure the time it took to complete the request.

Our evaluation mainly focuses on attacks in which the adversary infers sensitive information from the size of the resources that are returned to the victim when requesting specific endpoints. As such, we evaluate potential attack techniques under the assumption that an adversary can trigger the victim’s browser to send authenticated requests to arbitrarily chosen endpoints. This can be easily achieved by a moderately motivated attacker due to the plethora of methods that can be used to execute arbitrary JavaScript code in a cross-origin context (with regard to the target endpoint). For instance, an attacker can trick the user in visiting his website using phishing via e-mail or social networks [29], register a typosquatting domain [41], launch an advertising campaign where JavaScript code or an iframe containing the attacker’s web page is included [54], register a stale domain from which a JavaScript file is included [43], redirect insecure HTTP requests [37], ... Note that recent attacks on TLS also assume an attacker can execute JavaScript code in the victim’s browser [16, 1, 62].

3.1 Operations Involving Resources

By looking at their typical “lifetime”, we identified six different operations that involve resources, as shown in Figure 1. In the first step, a resource is generated at the

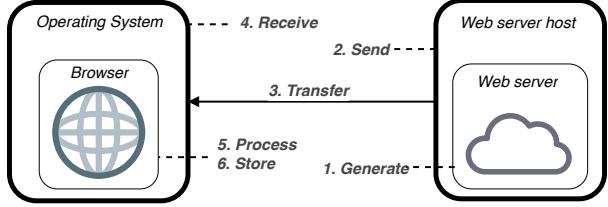


Figure 1: Overview of operations that involve resources.

side of the web server, as a result of the request initiated by the browser. Here the web server will associate the user’s state with the included cookie, produce the requested content, and pour it into an HTML structure. Although several attacks have been presented that can extract sensitive information from this generation process, e.g., direct timing attacks [7], our research focuses on methods that can expose the size of the generated content. Since the length of the response is only known *after* it has been generated, attacks against the resource generation process are excluded from our evaluation.

Once a resource has been dynamically generated, the machine where the web server is hosted on will send it back to the user that requested it. This means that if an adversary is able to observe the amount of traffic generated by the web server, he could use this information to infer the size of the response. We discuss size-exposing techniques in this context in more detail in Section 3.2.

When the resource leaves the web server, it is sent over several networks before it reaches the client. In any of these networks, an adversary capable of intercepting or passively observing the network traffic could be present. Because size-exposing attacks can be considered to be superfluous when an adversary can inspect the contents of a resource, we only consider encrypted traffic in our evaluation. Prior work has shown that popular encryption schemes such as SSL and TLS do not conceal the length of the original HTTP request and response, leading to various attacks [67, 12]. In our analysis, we extend this existing work by re-evaluating the feasibility of size-exposing attack methods when the new HTTP version (HTTP/2) is used. Furthermore, we explore possible size-exposing attack techniques in the context of Wi-Fi networks, where another layer of encryption is added, and elaborate on our findings in Section 3.3.

As soon as the response reaches the client’s machine, it is first received by the network interface, and then sent to the browser, where it is processed and possibly cached. Similar to the server-side, an adversary with a foothold in the operating system, can leverage traffic statistics to uncover the resource’s length. In Section 3.2, we investigate these types of attack techniques under various threat models, both for mobile devices as well as desktops.

Table 1: An overview of size-exposing attack techniques with their associated resource operations (as per Figure 1) and whether the techniques can be used to obtain the exact size of a resource.

Size-exposing technique	Resource operation	Exact size	References
Cache timing attacks	2, 4		[48, 76, 72, 44]
Traffic statistics pseudo-files	2, 4	✗	[77], Section 3.2
SSL/TLS traffic analysis	3	✗	[67], Section 3.3
Wi-Fi traffic analysis	3	✗	Section 3.3
Cross-site timing attacks	3		[7, 20]
Browser-based timing attacks	5		[60]
Storage side-channel leaks	6	✗	Section 3.4

After receiving the response, the browser will first signal the completion of the request by firing an Event. In the threat model we consider, the request is initiated by the malicious JavaScript code, and thus, its completion is signaled to the attacker. It is known that the time it takes for a request to complete is correlated with its size, giving rise to so-called timing attacks. However, these attacks have several limitations, and can only be used to obtain a rough estimate of a resource’s size. While a rough estimate is sufficient to perform certain attacks [7, 20], most of the real-world attacks we present in Section 4 require knowing the exact size of resources.

In a recent study, Van Goethem et al. found that the next step of a resource’s lifetime, i.e., parsing by the browser, is susceptible to timing attacks as well [60]. In contrast to classic timing attacks, these browser-based attacks do not suffer from network irregularities, and thus provide attackers with a more accurate and reliable estimate. Nevertheless, the maximum accuracy that can be achieved with these methods is still in the range of a few kilobytes, which is insufficient for some of the novel attacks presented in Section 4.

Finally, browsers may store resources in the cache, allowing them to be retrieved much faster in future visits. Motivated by the potentially nefarious consequences of caching resources chosen by an adversary, we analyzed the specification of the various APIs that are involved in this process. Surprisingly, we found multiple design flaws that allow an adversary to uncover the *exact* size of any resource. In Section 3.4, we elaborate in detail on these newly discovered vulnerabilities, and their presence in modern browsers.

An overview of all size-exposing techniques we discovered during our evaluation is provided in Table 1.

3.2 OS-based Techniques

In this section, we elaborate on size-exposing techniques that occur at the level of the operating system, on the side of the web server and client. In our analysis, we

considered four types of hosting environments for the web server, namely dedicated hosting, shared hosting, and cloud-based solutions (VMs and PaaS). To be able to observe the length of resources in the case of a dedicated hosting environment, an attacker would need to have either physical access, or infect the machine with a malicious binary. In both cases, we argue that the capabilities of the attacker far surpass what is required for a size-exposing attack, thereby making other attack vectors more appealing to the attacker.

The same argument applies to cloud-based hosting. It has been shown that cache-based side-channels attacks can extract sensitive information, including traffic information, in a cross-tenant or cross-VM environment [48, 76, 72]. However, if an attacker would have the capabilities to leverage a cache-based attack to accurately determine the size of a requested resource, this would mean that the attacker could also leverage the cache-based attack to determine (part of) the execution trace, which can be considered as significantly more severe in most scenarios. Given the lack of incentive for an attacker to uncover the resource size by launching a cross-tenant or cross-VM attack, we do not consider this in more detail.

In a shared hosting environment, web requests for several customers are served by the same system. Next to cross-process cache-based side-channel attacks, which can be considered similar to the above-mentioned cross-VM attacks, adversaries can typically also access the system-wide network statistics. These network statistics can be obtained by either running the `ifconfig` command, or by reading it directly from system pseudo-files such as `/proc/net/dev`. As these network statistics report the exact amount of bytes sent and received by a network interface, an adversary could leverage this information to uncover the size of a response. The attacker’s accuracy will of course depend on the amount of background traffic, but the ability to coordinate with the victim’s browser gives the adversary a strong advantage. Because shared hosting environments are typically used by less popular websites, we consider this type of attack scenario to be unlikely, and thus do not explore this issue further.

On the side of the client, we explored various size-exposing techniques, but found that most techniques either require too many privileges, e.g., infecting the system with a malicious binary, or yield inaccurate results [44]. An interesting exception is the Android operating system, which also keeps track of network statistics. In addition to the global network statistics, Android also exposes network statistics *per user*, which, surprisingly, can be read out by any application without requiring

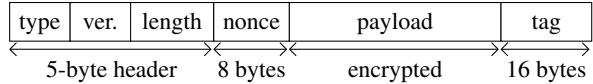


Figure 2: TLS record layout when using AES-GCM.

ing permissions¹. In their work, Zhou et al. showed that by passively monitoring network statistics on Android, an adversary can infer sensitive information from the requests made by other applications. We make the observation that these attacks can be extended when considering an attacker model in which the adversary can actively trigger specific requests in the victim’s mobile browser. As a proof-of-concept application, we created an HTTP service, which reports the number of bytes received by the user associated with the `com.android.chrome` application. Finally, our application triggers the mobile browser to open a web page, which first contacts the local service, next downloads an external resource, and then obtains the network statistics again, allowing us to determine the exact size of the external resource.

3.3 Network-based Techniques

We now show the size of a resource can be uncovered by monitoring its transmission over a secure connection. First we do this for TLS, and then we evaluate the case where Wi-Fi encryption is used on top of TLS. Although Wi-Fi hides individual connections, effectively offering a secure channel similar to that of VPNs or SSH tunnels, we show attacks remain possible. We also study the impact of the new HTTP/2 protocol.

3.3.1 Transport Layer Security (SSL / TLS)

Web traffic can be protected by HTTPS, i.e., by sending HTTP messages over TLS [47, 15]. Once the TLS handshake is completed, TLS records of type application data are used to send HTTP messages. The type and length of a record is not encrypted, and padding may be added if block ciphers are used. Since nowadays more than half of all TLS connections use AES in Galois Counter Mode (GCM) [27], we will assume this cipher is used unless mentioned otherwise. The layout of a TLS record using AES-GCM is shown in Fig. 2. Note that for this cipher no padding is used. An HTTP message can be spread out over multiple TLS records, and in turn a TLS record can be spread out over several TCP packets. An endpoint can freely decide in how many records to divide the data being transmitted.

¹These statistics can be read out from the pseudo-files `/proc/uid_stat/[uid]/tcp_rcv`, or, since Android 4.3, can be obtained from the `getUidRxBytes()` interface.

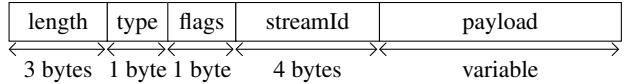


Figure 3: Simplified HTTP/2 frame layout.

To determine the length of a resource sent over TLS, we first need to know when it is being transmitted. We accomplish this by using JavaScript to make the victim’s browser fetch a page on our server, signaling that the next request will be to the targeted resource. We then monitor any TLS connections to the server hosting this resource, which is possible because the TCP/IP headers of a TLS connection are not encrypted. Once the resource has been received, we again signal this to our server. This enables us to identify the (single) TLS connection that was used to transmit the resource. Finally we subtract the overhead of the TLS records (see Figure 2) to determine the length of the HTTP response. If the connection uses a cipher that does not require padding, this reveals the precise length of the HTTP response. Otherwise only a close estimate of the response length can be made. By subtracting the length of the headers from this HTTP response, whose value can be easily predicted, we learn the length of the requested resource.

We tested this attack against two popular web servers: Apache and nginx. Even when the victim was actively browsing YouTube and downloading torrents, our attack correctly determined the length of the resource. Interestingly, we noticed that Apache puts the header of an HTTP response in a single, separate, TLS record. This makes it trivial to determine the length of the HTTP response header sent by Apache: it corresponds exactly to the first TLS record sent by the server.

We also studied the impact of the HTTP/2 protocol [4] on our attacks. HTTP/2 does not change the semantics of HTTP messages, but optimizes their transport. In HTTP/2, each HTTP request and response pair is sent in a unique stream, and multiple parallel streams can be initiated in a single TCP connection. The basic transmission unit of a stream is a *frame* (see Figure 3). Each frame has a *streamId* field that identifies the stream it belongs to. Several types of frames exist, with the two most common being header and data frames. Header frames encode and compress HTTP headers using HPACK [45], and data frames contain the body of HTTP messages. Nearly all other frames are used for management purposes, and we refer to them as control frames. Most browsers only support HTTP/2 over TLS. Usage of HTTP/2 is negotiated using the Application Layer Protocol Negotiation (ALPN) extension of TLS. This extension is sent unencrypted, meaning we can easily detect if a connection uses HTTP/2.

To determine the size of a resource transmitted using HTTP/2 over TLS, we have to predict the total overhead created by the 9-byte frame header (see Figure 3). Moreover, we need to be able to filter away control frames. Both Apache and nginx send control frames in separate TLS records, and these records can be detected by their length and position in the TLS connection, allowing us to recognize and filter these frames. To calculate the overhead created by the 9-byte frame header, we need to predict the number of HTTP/2 data frames that were used to transmit the resource. For Apache this is easy since it always sends data frames with a payload of 2^{14} bytes, except for the last frame. For nginx, the number of data frames can be predicted based on the number of TLS records. This means that for both servers we can predict the amount of overhead HTTP/2 introduces. The size of the HTTP/2 header frame can be predicted similar to the HTTP/1.1 case, with the addition that the HPACK compression has to be taken into account. Finally, we found that multiple streams are active in one TCP connection only when loading a page. By waiting until the HTTP/2 connection is idle before letting the victim’s browser fetch the resource, the only active stream will be the one downloading the resource. All combined, these techniques allowed us to accurately predict the size of resources sent using HTTP/2. Note that if the server uses gzip, deflate, or similar, we learn the compressed size of the resource. In Section 4, we show that this is sufficient to perform attacks, and can even be used to extend an attacker’s capabilities.

3.3.2 Encrypted Wi-Fi Networks

Wireless networks are an attractive target for traffic monitoring attacks. For instance, our attack against TLS can be directly applied against open wireless networks. However, these days many wireless networks are protected using WPA2 [71]. This means that all packets, including their IP and TCP headers, are encrypted. Hence we can no longer use these headers to isolate and inspect TLS connections. Nevertheless, we show it is possible to uncover the size of an HTTP message even when Wi-Fi encryption is used on top of TLS.

In the Wi-Fi protocol, the sender first prepends a fixed-length header to the packet being transmitted, and then encrypts the resulting packet [28]. To encrypt and protect a packet, the only available ciphers in a Wi-Fi network are WEP, TKIP, or CCMP. Note that WPA1 and WPA2 are not ciphers, but certification programs by the Wi-Fi Alliance, and these programs mandate support for either TKIP or CCMP, respectively. Since both WEP and TKIP use RC4, and CCMP uses AES in counter mode, padding is never added when encrypting a packet. Therefore, no matter which cipher is used, we can always determine the

precise length of the encrypted plaintext. Finally, Wi-Fi encryption is self-synchronizing, meaning that a receiver can decrypt packets even if previous ones were missed or blocked.

Similar to our attack against TLS, we determine when the resource is being transmitted by signaling our own server before and after we fetch the targeted resource. However, we can no longer easily determine which packets correspond to the requested resource as Wi-Fi encrypts the IP and TCP headers. Consequently, any background traffic will interfere with our attack. One option is to execute the attack only if there is no background traffic. Unfortunately, if the user is actively browsing websites or streaming videos, periods without traffic are generally too short. In other words, it is hard to predict whether a period without traffic will be long enough to fetch the complete resource. Our solution is to wait for a small traffic pause, and extend this pause by blocking all packets that are not part of the TCP connection that will fetch the resource. Blocking packets in a secure Wi-Fi network is possible by using a channel-based man-in-the-middle (MitM) attack [61]. Essentially, the attacker clones the Access Point (AP) on a different channel, and forwards or blocks packets to, and from, the real AP. The channel-based MitM also has another advantage: if the adversary misses a packet sent by either a client or AP, the sender will retransmit the packet. This is because the cloned AP, and cloned clients, must explicitly acknowledge packets. Hence our attack is immune to packet loss at the Wi-Fi layer. Once we start measuring the size of the resource, we only forward packets that could be part of the connection fetching this resource. First, this means allowing any packets with a size equal to a TCP SYN or ACK. Second, we have to allow the initial TLS handshake and the HTTP request that fetches the resource. Since both can be detected based on the length of Wi-Fi packets, it is possible to only forward packets that belong to the first TLS handshake and HTTP request. By blocking other outgoing requests, servers will refrain from replying with new traffic. Hence we can still fetch our targeted resource, but all other traffic is temporarily halted.

In experiments the above technique proved highly successful. Even when the victim was browsing websites or streaming YouTube videos, it correctly isolated the TLS connection fetching the resource. We also tested the attack when the victim was constantly generating traffic by sending ping requests of random sizes. Since the size of these packets rarely matches that of a TCP ACK/SYN or TLS handshake packet, all ping requests were blocked, and the correct connection was still successfully isolated.

The next step is to subtract the overhead added by Wi-Fi and TLS. Since none of the cipher suites in Wi-Fi use padding, it is straightforward to remove padding added by the Wi-Fi layer. However, we cannot count the

number of TLS records sent as their headers are now encrypted. Nevertheless, for both nginx and Apache with HTTP/1.1, we found that a new TLS record is used for every 2^{14} bytes of plaintext. This allows us to predict the number of TLS records that were used, and thereby the overhead created by these records. We discovered only one exception to this rule. If an Apache server uses chunked content encoding, each chunk is sent in a separate TLS record. This means that the number of TLS records become application-specific, and the attacker has to fine-tune his prediction for every targeted resource. We remark that this behavior of Apache is not recommended, because it facilitates chunked-body-truncation attacks against browsers [5].

When HTTP/2 is used, the situation becomes more tedious. Here we have to predict both the number of TLS records, as well as the number, and types, of HTTP/2 frames. We found that these numbers are predictable for the first HTTP/2 response in a TLS connection. Since all browsers limit the number of open TCP connections, we first close existing connections by requesting several pages hosted on different domains. After doing this, a new connection will be used to fetch the targeted resource, meaning we can predict the amount of overhead. Apache always uses HTTP/2 data frames with a payload of 16348 bytes, even when chunked content encoding is used. Furthermore, the TLS records always have a payload length of 1324, except for every 100th TLS record, which has a length of 296. Finally, Apache always sends the same three HTTP/2 control frames, spread over two TLS records, before sending the resource itself.

For new TLS connections, nginx sends three initial HTTP/2 control frames in either one or two TLS records, where most of the time only one TLS record is used. Then it enters an initialization phase where the first 10 TLS records have a predictable size, with each size taken from the set {8279, 8217, 4121, 4129}. After this initial phase, it repeats the sequence [16408, 16408, 16408, 16408, 96], with the exception that at relatively infrequent and random times a TLS record of size 60 is used instead of 96. However, as this is only a small difference, it generally affects the number of TLS records by at most one. All combined, if we assume the least number of TLS records are used, we underestimate the actual number of TLS records by at most two. In fact, most of the time no extra records are used. Hence an attacker can make multiple measurements, and pick the most common length as being the one without the extra (one or two) records.

3.4 Browser-based Techniques

Over the last few years, one of the most important evolutions on the web is the increase of support for mobile

Algorithm 1 Uncover the size of resources by abusing the per-site quota limit

```

response ← fetch(url)
fillStorage()
size ← 0
loop
    freeByteFromCache()
    size ← size + 1
    storageResult ← cache.put(response)
    if storageResult == True then
        return size
    end if
end loop

```

devices. This advancement requires that all the characteristics that are specific to mobile devices are properly accommodated. For instance, mobile devices travel along with their users, which means that every now and then the devices become disconnected, preventing the user from accessing any web-based content. Recent advancements in browser design aim to tackle this problem with a promising API named ServiceWorker [66]. The core idea behind the ServiceWorker API is to allow websites to gracefully handle offline situations for their users. For example, a news website might download and temporarily store news articles when users are connected, allowing them to still access these while being disconnected. Note that although we mainly focus on the ServiceWorker API, all attacks can also be applied by using ApplicationCache [63], the caching mechanism that ServiceWorker aims to replace.

3.4.1 Per-site quota

For caching operations, the ServiceWorker API provides a specific set of interfaces, named Cache API, which can be used to store, retrieve and delete resources. A noteworthy aspect of the Cache API is that it allows one to cache any resource, including cross-origin responses. Furthermore, to limit misuse cases where a malicious player takes up all available space, the per-site² storage is restricted. This restriction is shared among a few other browser features that allow persistent data storage, for instance localStorage and IndexedDB. The way per-site quota is applied, is decided by the browser vendor; for the most popular browsers this is either a fixed value in the range of 200MB to 2GB, or a percentage - typically 20% - of the global storage quota [22, 42, 32].

For the purpose of exposing the size of resources, having full control over the cache, and the fact that this cache

²According to the current specification of the Storage API, a site is defined as eTLD+1, meaning `foo.example.org` and `bar.example.org` belong to the same site, whereas `foo.host.com` belongs to a different site [70].

is limited by a fixed quota, are two very interesting aspects. An adversary can directly leverage these two features to expose the size of any resource by means of the pseudo-code listed in Algorithm 1. In the attack, the resource is first downloaded using the Fetch API, which will result in an "opaque" Response. Next, the adversary makes sure that the site's available storage is filled up to the quota. In practice, we found that by storing large data blobs using the IndexedDB API, the storage speed approaches the maximum writing speed of the hard disk, allowing the attacker to reach the quota in a few seconds. In a final step, the adversary will free up one byte from the cache and attempt to store the response. This storage attempt will only succeed if sufficient quota is available, otherwise more bytes should be freed. Eventually, the attacker learns the exact size of the resource by the number of bytes that were freed until the resource could be stored. Note that the resource only needs to be downloaded once, resulting in a significant speed-up of the attack. In our experimental setup, the initial attack could be executed in less than 20 seconds, and subsequent size-exposing attempts were performed in less than a second as the quota had already been reached.

3.4.2 Global quota

In addition to the storage restrictions of sites, browsers also enforce a global storage quota to ensure normal system operations are not affected. When this global quota is exceeded, the storage operation will not be canceled, but instead the storage of the least-recently used site will be removed. As a result, the two features required to expose the size of a resource, i.e., full control over the cache and an indication when the quota is exceeded, are present. In comparison to the size-exposing attack that leverages the per-site quota, this vulnerability is considerably harder to successfully exploit: the attacker needs to reach the global quota limit, which needs to be spread over multiple sites, and has to take into account that the global quota can fluctuate as a result of unrelated system operations. Nevertheless, for the purpose of creating an improved design, it is important to consider all flaws of the current system. Furthermore, on systems with a limited storage capacity, e.g., mobile devices, some of these restrictions may not apply, increasing the feasibility of an attack.

A simplified, unoptimized method that can be used to expose the size of an arbitrary resource is provided in Algorithm 2. Similar to the per-site quota attack, the adversary first downloads the resource and temporarily stores it in a variable. Next, a site is filled with a certain amount of bytes ($storageAmount$) which should be larger than the size of the resource. In a following step, the adversary will need to fill the complete quota. Since

Algorithm 2 Uncover the size of resources by abusing the global quota limit

```

response ← fetch(url)
storageAmount ← 5MB
site0.addBytes(storageAmount)
i ← 1
while !isEvicted(site0) do
    storageResult ← sitei.addBytes(1)
    if storageResult != True then
        i ← i + 1
    end if
end while
site0.cache.put(response)
remainingBytes ← 0
while !isEvicted(site1) do
    site0.addBytes(1)
    remainingBytes ← remainingBytes + 1
end while
size ← storageAmount – remainingBytes

```

for most major browsers, the global quota is set to 50% of the total available space on the device, and the per-site quota is set to either a percentage of the global quota or a fixed size, the adversary will need to divide this over multiple domains. As soon as the eviction of the first site is triggered, the adversary knows the exact amount of freed space, namely $storageAmount$. Finally, the adversary adds the resource to an empty site and fills it until the global quota is reached again, which can be observed by checking for the eviction of the next least-recently used site, i.e., $site_1$. The size of the resource can then be calculated as the original size of the first site subtracted by the number bytes required to reach the global quota again ($remainingBytes$).

3.4.3 Quota Management API & Storage API

The last attack involving browser storage abuses the Quota Management API [65], and the similar Storage API [70]. These APIs aim to give web developers more insight into their website's storage properties, more specifically the number of bytes that have been stored and the space that is still available. At the time of writing, the Storage API is still being designed, and will consolidate the storage behavior of all browsers into one agreed-upon standard.

The functionality provided by the Quota Management API is the direct source of a size-exposing vulnerability that is worryingly trivial to exploit. An adversary can simply request the current storage usage, add a resource to the cache, and retrieve the storage usage again. Since the Quota Management API will return the usage in bytes, the *exact* resource size can be obtained by subtracting the two usage values. Although the Quota

Management API has only been adopted by the Google Chrome browser, this browser alone accounts for approximately 48% of the market share [56], leaving hundreds of millions of internet users vulnerable to this highly trivial size-exposing attack vector. Despite our efforts of reporting these findings to the Chrome team, all up-to-date versions of the Google Chrome browser remain allowing this API to be used by any website, without the user’s knowledge.

Because the per-site quota is related to the global quota³, the Quota Management API can also be used to infer the caching operations of a different website. For instance, a malicious iframe that is embedded on a website could observe changes in the available quota, and infer the length of cached resources. This information could in turn be used to either analyze the interactions of the user on the website, or disclose private information based on the length of the cached resources. A similar attack scenario is discussed in more detail in Section 4.4. Another interesting case occurs when making the observation that the per-site quota is also related to the total free disk space. The byproduct of this behavior is that an adversary can also observe the disk operations of other, possibly security-sensitive, processes. As this issue is unrelated to size-exposing techniques, we do not explore this vulnerability in more detail.

The functionalities provided by the Quota Management API are directly responsible for the vulnerabilities discussed in this section. It is unclear why this API was developed without taking into account potential security and privacy implications. In essence, these findings serve as a strong indicator that new browser features should be thoroughly reviewed for security and privacy flaws. Since the Storage API provides the same functionality as the Quota Management API, the same issues arise there as well. At the time of writing, the Storage Standard deviates from the Quota Management API in the sense that it states that a “rough estimate” should be returned. Because the term “rough estimate” is not formally defined, implementations of this specification are likely to still be vulnerable to statistical attacks, as the quota limit can easily be requested thousands of times. In Section 5.1 we propose a new API design that protects against all browser-based size-exposing techniques we discussed in this paper.

4 Real-world Consequences

In contrast to prior work on size-exposing techniques, which is mainly focused on passive network observation, the attacks presented in this paper leverage the abil-

ity to request arbitrarily chosen resources in the victim’s browser. To provide more insight into the consequences and potential attack scenarios, we explore a selection of real-world cases where one of the size-exposing techniques can be used to extract private and sensitive information from the victim. The list of attacks that are discussed, is by no means the exclusive list of possible targets. Instead, we made a selection of attack scenarios to provide a variety in methodology, type of disclosed information, and category of web service.

Ethical Considerations To evaluate the severity and impact of size-exposing techniques on internet users, it cannot be avoided to evaluate these attacks on real-world services. To prevent any nefarious consequences of this evaluation, all attacks were manually tested, and were performed exclusively against our own accounts. As a result, from the perspective of the tested services our analysis only generated a restricted amount of legitimate traffic. Moreover, users of the analyzed websites were not directly involved in our attacks. For the quantitative case-studies, we only obtained publicly available information, and present it in anonymized form. Given the above-mentioned precautions, we believe our evaluation of real-world services did not have any adverse effects on the tested subjects.

4.1 User Identification

Virtually every online social network provides its users with their own profile page. Depending on the user’s privacy settings, these profile pages typically are completely or partially available to anyone. In the attack scenario where the adversary is interested in learning the identity of the victim, the adversary first collects the publicly available data from (a subset of) the users of the social network. Later, during the actual size-exposing attack, he tries to associate the data obtained from the victim to a single entry from the public data, allowing him to expose the victim’s identity. To evaluate the feasibility in a real-world environment, we exemplify the attack scenario on Twitter, one of the largest social networks.

By default, the profile of each Twitter user is public, and contains information on the latest tweets that were created by the user, the list of followers and followees, the tweets that were “liked” by the user, and the lists he/she follows and is a member of. Except for the user’s tweets, each type of information can be accessed by a link that is shared by all Twitter users, e.g., the page located at <https://twitter.com/followers> lists the last 18 accounts that follow the user. For each follower, the name, account name and short biography is shown.

The main assumption in this attack scenario is that the combined length of all parts that constitute to the

³The per-site quota is 20% of the global quota in Google Chrome; for Firefox this is the case as well when the disk space is less than 20GB.

resource, i.e., the names, account names and bios of the last 18 followers, is relatively unique. To validate this assumption, we performed an experiment that reflects an adversary’s actions in an actual attack scenario. For this experiment, we obtained publicly available information of 500,000 users, which were selected at random from the directory of public profiles provided by Twitter⁴. More specifically, we downloaded the resources located at `/following`, `/followers`, `/likes`, `/lists` and `/memberships`, and recorded the associated resource size, both with and without gzip compression.

Next, we grouped together Twitter accounts that share the same resource length, e.g., if the `/following` resource is 281026 bytes for only two users, these users form a group of size 2. In Figure 4 we show the percentage of Twitter accounts for all group sizes, for the compressed and uncompressed resource size. Note that a logarithmic scale is used for the percentage of Twitter accounts on the y-axis. This graph clearly shows that when the size of multiple resources is combined, the majority of Twitter accounts can be uniquely identified. By exposing the size of the uncompressed `/following` and `/followers` resources, 89.66% of the 500,000 Twitter accounts can be uniquely identified. When the size of all five resources is known, the identity of 97.62% of the Twitter accounts can immediately be uncovered. The graph also clearly shows that when gzip compression is applied, the group sizes of individual resources becomes larger, which is most likely due to the reduction in entropy of resource sizes. Nevertheless, when the size of multiple compressed resources are combined, a uniqueness comparable to the size of uncompressed resources is achieved: 81.69% Twitter accounts can be uniquely identified when the size of the `/following` and `/followers` resources is combined; for all five resources, this is 99.96%. The most likely explanation for this is that in case a resource is virtually empty, i.e., the account name is the only dynamic part of the resource, not only the length but also the content of the account name is reflected in the compressed resource size.

Although the viability of this attack was only evaluated on a subset of all Twitter accounts⁵, this experiment does suggest that adversaries can immensely narrow down the number of possible candidates for the user’s identity by knowing the size of just five resources. Furthermore, various techniques exist that can uniquely identify a user among a limited set of accounts [33, 26], making user-identification by exposing the size of resources well within the reach of a moderately motivated attacker.

⁴<https://twitter.com/i/directory>

⁵Twitter has approximately 320 million active accounts.

4.2 Revealing Private Information

Next to revealing the identity of a web user, adversaries may also be interested in learning private information. A particular type of information that, in general, is considered highly sensitive, is information concerning medical conditions. To evaluate whether our novel size-exposing techniques can be used to also disclose this type of data, we explored the performance of such techniques on WebMD, one of the leading health information services websites. One of the features provided by WebMD is “Health Record”, a web service that allows users to organize their personal health records⁶. More precisely, users can add, and keep track of, their medical conditions, medications, allergies, etc. For each entry, the user can choose among an exhaustive list of terms. For instance, there are 4,105 different medical conditions that can be selected.

At any point in time, users can download their own medical report, either as automatically generated PDF or in plain text format. It should be noted that the types of medical records that are shown in this report is specified by the user (or attacker), and that the PDF is sent without compression, whereas the textual report is served with gzip compression. Although there is some variety in the length of the possible terms, it is insufficient for an adversary to determine which medical conditions the user suffers from: on average, a certain length is shared among 124.59 possible medical conditions. However, if the adversary can obtain the resource size both with and without compression, this can significantly improve his attack: in this case, the group size can be limited to 35.50 on average. This can be achieved by various methods, e.g., by obtaining the length from two resources that share the same content, where one is served with compression and the other without, or by tricking the server in sending the resource without compression⁷, or even by combining the browser-based attacks with the network-based attacks. In case the sensitive content is present on multiple compressed resources (in this case, this can be triggered by varying the types of medical records that are reported), the group size can be reduced even further. In the attack scenario against WebMD, a single iteration of this technique, i.e., including the medical condition on a compressed resource with other known content, reduces the average group size to 18.73. By applying multiple iterations, each with slightly different content, it becomes possible to uniquely identify the user’s medical condition in most cases.

⁶<https://healthmanager.webmd.com/>

⁷When a resource is included as a `<video>` element, the `Accept-Encoding` header will be either absent or set to `identity`, causing most web servers to send it without compression.

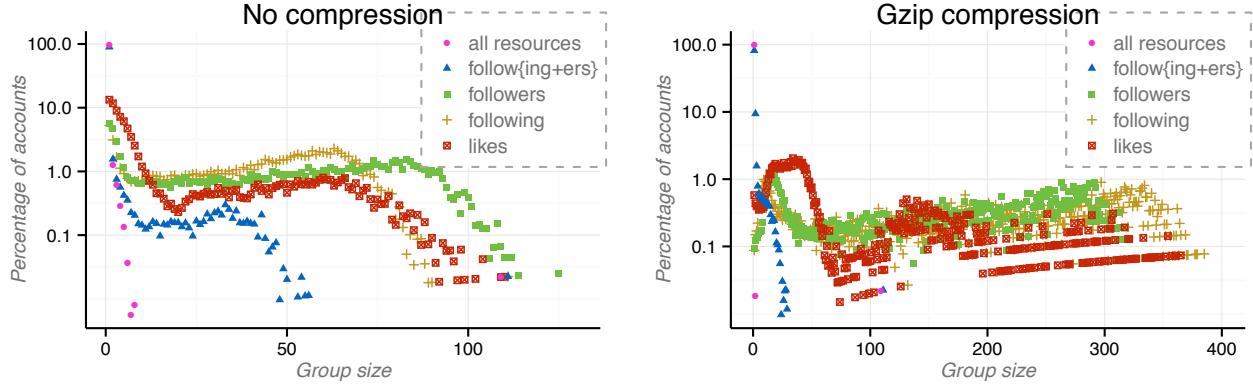


Figure 4: Percentage of Twitter accounts that share the same resource length with a group of varying size.

4.3 Search-Oriented Information Leakage

Many web applications allow their users to search the data they (in)directly entered. For instance, web-based e-mail clients provide the functionality to search for certain messages. In a recent study, Gelernter et al. show that this functionality can be abused by attackers to disclose sensitive information, such as the user’s identity and credit-card numbers [20]. In their attacks, the researchers leverage the fact that in certain cases query parameters are reflected in the results. Consequently, when a search query has several matches, the resulting resource size will be considerably larger than with an empty result-set, allowing an adversary to resort to timing attacks to determine whether a certain search query yielded results. Several service providers that were shown to be vulnerable to these attacks implemented a mitigation by preventing query parameters to be reflected in the search results. Although these measures effectively thwart the above-mentioned attacks, the web services remain vulnerable to the size-exposing attacks proposed in this paper, as these disclose the size of a resource with 1-byte precision.

4.4 Cross-Origin Cache Operations

Telegram is a popular cloud-based instant messaging service, particularly known for its security and encryption features. Not surprisingly, these features have attracted terrorist organizations to use the service as a secure communication channel [53]. This, in turn, makes Telegram a valuable target for intelligence agencies to find members of terrorist groups. Since all exchanged messages are encrypted using MTProto, which was shown to only suffer from minor theoretical attacks, plaintext-recovery is considered to be unlikely [30].

Next to the mobile and desktop versions of the Telegram application, a web-based version is provided as well⁸. An interesting feature of this web-based version is that when a photo is shared in a group, the web appli-

cation will use the File API [46] to cache two thumbnails of the photo. Because the storage used by the File API counts towards the global cache quota, it is possible to infer whether a resource is being cached as per the attacks discussed in Section 3.4.2 and Section 3.4.3.

In an attack scenario where the adversary tries to determine group membership of the victim, the attacker first lures the victim to his malicious web page. On this web page, the adversary includes the page of the target group in an iframe. Telegram does not use the X-Frame-Options header, but instead makes the content invisible by default through CSS, and uses JavaScript to make it visible in case no framing is detected (a popular Clickjacking defense proposed by Rydstedt et al. [51]). As a result, the page’s content will be loaded, but remains invisible, and impossible to interact with⁹. If the user is member of the targeted group, the Telegram website will download and cache thumbnails of the latest media items that have been shared in the group, resulting in a change of the available quota. Otherwise, a message is shown stating that the user is not a member of the group. As an additional verification step, the adversary could post another photo in the group, and witness a change in the available quota. By leveraging our novel size-exposing techniques, we found it was trivial to detect group membership. Because the MTProto scheme only provides very limited padding, group membership can also easily be detected by analyzing the size of HTTP responses.

5 Defense Mechanisms

In this section we discuss various mechanisms that can be used to thwart size-exposing attacks. Due to space limitations, we only focus on a limited set of defense mechanisms, which were selected on the basis of completeness, novelty, amount of overhead and ease of adoption.

⁸The <iframe> element should have a sandbox attribute set to "allow-scripts allow-same-origin" to prevent top level navigation, while ensuring the page is loaded properly.

⁹<https://web.telegram.org>

5.1 Hardening Browser Storage

As was shown in Section 3.4, several features related to the storage operations in browsers can be abused to expose the size of cross-origin resources. At the time of writing, there exists no universal specification that standardizes these operations. However, the Storage API specification is being developed with the purpose of designing a unified definition that will be adopted by all browsers. In its current state, the Storage API consolidates the current browsers behavior regarding the quota limit per website. Furthermore, it incorporates the functionalities offered by the Quota Management API.

We propose a countermeasure that extends the Storage API. To make adoption by browsers feasible, we aim to provide a usable solution, i.e., normal application behavior should not be jeopardized. As a result of the feedback provided by the communication with specification editors and browser vendors, we opted for an approach where “virtual padding” is applied to resources. To prevent an adversary from learning the size of a resource, either by abusing the storage limit or by requesting the available quota, this size should be masked with a random value. However, it is a well-known fact that by adding a random value, the mechanism becomes subject to statistical attacks. Because resources can be added to the cache extremely fast, an adversary is able to obtain a large number of observations in a limited amount of time, putting him in a very strong position.

Inspired by a mitigation for web-based timing side-channels proposed by Schinzel [52], and by making the observation that in contrast to caching operations, downloading a resource takes a considerable amount of time, we propose the following defense. When a resource is downloaded as the result of a `fetch()` operation, we associate a unique identifier, `uid`, with the `Response` object. Next, we compute $q = \lceil \text{size} + \text{hash}(\text{secret} + \text{uid}) \rceil_{\Delta}$, where `size` is the size of the resource, `hash()` a uniformly distributed hash function yielding integers in the range $[0, p_{max}]$, and `secret` a cryptographic random number that is associated to a single browsing session¹⁰. The total size q is then rounded up towards the nearest multiple of Δ to prevent an attacker from learning the bounds of the added padding. When the `Response` is added to the cache, the per-site and global quota will be increased by q . This value should also be stored as part of the `Response` object to ensure that for each cache operation the same value is either added or subtracted from the quota. As a result, the only way for an adversary to obtain a new observation is to download the same resource again. It should be noted that the padding that is added for each cache operation is virtual, in the sense that these

¹⁰To prevent an adversary from linking two browser sessions, `secret` is changed whenever the browser session changes.

bytes are not actually written to the disk, but are just kept as a type of bookkeeping.

It is clear that the overhead on the quota and the security guarantees provided by this defense method are directly related to the values of p_{max} and Δ . In fact, this provides a trade-off between security/privacy and usability, for instance, the larger the value of p_{max} , the harder it will be for an adversary to uncover the size of resources (within certain boundaries), but on the other hand, a large p_{max} will entail a smaller storage capacity due to the amount of padding. We argue that with an analysis on the typical use-cases of caching operations, these values could be defined to accommodate legitimate behavior while preventing attacks. Furthermore, it could be taken into account that this mechanism generates a virtual loss in storage capacity, and therefore the quota could be increased to account for this. In addition, it is possible to apply a rate-limiting approach to limit the amount of observations that can be made by an adversary. For instance, if the reported quota is only updated once every minute, statistical attacks can be largely mitigated, which in turn allows for smaller values of p_{max} , and restricts the (already virtual) overhead.

Given the generality of the defense, its strong security guarantees, and the low overhead, we feel confident that this approach, or a similar derivative thereof, will be incorporated into the HTML specification, and encourage browser vendors to mitigate the attacks presented in Section 3.4 in this manner.

5.2 Detecting Illicit Requests

In essence, the size-exposing techniques presented in this paper require the ability to initiate authenticated cross-origin requests, and rely on the targeted web service to handle the request in the same way it would for legitimate requests. This means that when either part is removed, i.e., either authenticated cross-origin requests are disabled, or the web server answers with a static error message, the complete class of size-exposing techniques will be mitigated. To accomplish this, it is possible to resort to existing, and well-established techniques in related research fields. For instance, by blocking third-party cookies, which is typically used to prevent tracking on the web [50], the cross-origin requests initiated by the adversary will be sent without the cookie. As a result, the website will handle the request as if the user was not logged in, preventing the adversary from learning anything about the user’s state at the website. Mozilla and the Tor Browser project are working on minimizing the limitations imposed by blocking third-party cookies, by implementing a feature name double-keyed cookies, which binds cookies to the origin pair (first-party, third-party), and aims to prevent the risks of breaking sites

caused by blocking cookies [9, 59]. Similarly, certain browsers provide the ability to attach third-party cookies only if these were set during top-level navigation, and block these otherwise. While this technique can be used to prevent tracking by unknown parties, it does not adequately prevent the attacks presented in this paper as the targeted third-party services are the ones that are actually used by the victim.

On the side of the server, solutions similar to those that prevent Cross-Site Request Forgery (CSRF) attacks could be applied. A well-known method, as proposed by Barth et al., to accomplish this, is to analyze the `Origin` and/or `Referer` headers and only allow requests from trusted origins [2].

5.3 Network-based Countermeasures

Padding can be used to hide the length of resources during their transmission. Since general-purpose padding schemes are already well-studied, we do not discuss them further. Instead, we focus on countermeasures that fit our use-case, where only the size of sensitive dynamically generated resources must be protected. This allows us to provide a countermeasure with low overhead and high security guarantees, at the cost of requiring some effort on the web administrator’s part.

Our idea is to add an amount of padding based on the hash of the session cookie, the URL, and any parameters that affect the generation of the resource. More formally, $\text{padding} = \text{hash}(\text{cookie} + \text{url} + \text{params})$. If the user is not logged in, no padding is added. For each resource, the parameters that influence the generation of the resource must be manually specified. Other parameters should not be included, otherwise an adversary can add bogus parameters to obtain a new padding value for the same resource. This construction assures that sensitive resources, for any specific user, receive an amount of padding that is unpredictable by an attacker. However, this padding remains identical over several requests, meaning it even guarantees protection against statistical attacks. Information can only be leaked if the resource changes over time. This can happen when the attacker was able to affect the generation of the resource on the server, or simply because the information contained in the resource has changed over time. In this situation an observer can learn the difference in resource size. If the resource does not contain variable content, such as dynamic advertisements, this attack can be mitigated by including the content of the resource in the hash function. Similar to hardening the browser (see Section 5.1), the security guarantees depend on the value of p_{\max} . Provided the hash function is uniformly distributed, this countermeasure introduces on average $\frac{p_{\max}}{2}$ bytes of overhead.

For wireless networks, where we assume Wi-Fi encryption is used on top of TLS, we can rely on the previously mentioned techniques to protect the TLS connection. Additionally, an identifier-free wireless protocol can be used, making it more difficult for an attacker to attribute Wi-Fi packets to specific clients [23, 18, 3, 8].

6 Related Work

Size-exposing techniques have surfaced in several research areas, ranging from timing attacks, to network traffic analysis, to browser-based and cross-VM side-channel leaks. As part of an in-depth analysis, which lead to the discovery of multiple novel attack methods, we already touched upon a variety of related work, as discussed in Section 3. In this section, we give a brief overview of the most relevant work, and discuss it in the context of our findings.

Prior research that analyses methods that can expose the size of an attacker-specified resource, is mainly focused on leveraging timing as a side-channel information leak [19, 7, 14, 20, 60]. Because timing attacks measure the time required to download or process a resource, which is often influenced by various factors such as network irregularities or background noise, these attacks have certain limitations with regards to the accuracy of the uncovered resource size. In our research, we presented novel techniques that leverage the browser-imposed quota to reveal the *exact* size of any resource.

An interesting class of vulnerabilities where the size of resources is exploited, are compression side-channel attacks [31]. These attacks generally leverage the compression rate that is achieved when compressing an unknown value in a larger corpus of known values, allowing an adversary to uncover information about the unknown value from the resource size after compression. More recently, researchers have shown how similar attacks can be applied to various compression mechanisms used on the web [49, 21].

In the context of privacy-violating cross-origin attacks, Lee et al. have shown that the `ApplicationCache` mechanism can be used to uncover the status code that is returned for cross-origin resources [34]. Their attack exploits certain intricacies of `ApplicationCache`, which exhibits a different behavior based on the returned status code of referenced endpoints. The researchers did not explore vulnerabilities originating from the imposed quota and storage limits. Another type of attack that violates the principle of Same-Origin Policy is Cross-Site Script Inclusion (XSSI), first introduced by Grossman in 2006 [24], and recently analyzed on a wide scale by Lekies et al. [35]. In XSSI attacks, a dynamically generated JavaScript (or CSV [58]) file from a vulnerable website is included as a `<script>` element on the web

page of the attacker. The often sensitive content that is present in these files can then be obtained out by the adversary as a result of the modifications the script makes to the attacker-controlled DOM.

Compared to prior work on the analysis of web traffic [57, 6, 12, 36, 11, 10, 17], our work is, to the best of our knowledge, the first to combine traffic analysis with the ability to execute code in the victim’s browser. Similarly, traffic analysis works on Wi-Fi also assume a passive, instead of an active, adversary [8, 23, 3, 75, 74]. That is, we believe our work is the first to actively block specific Wi-Fi packets in order to measure the size of HTTP messages.

7 Conclusion

The size of resources can be used to infer sensitive information from users at a large number of web services. In our research, we performed an extensive analysis on the various operations that are performed on resources. As a result of this evaluation, we identified several new techniques that can be used to uncover the size of any resource. In particular, an attack that abuses the storage quota imposed by browsers, as well as a novel technique against Wi-Fi networks that can be used to disclose the size of the response associated with an attacker-initiated request. To provide more insight into how these attack methods can be applied in real-world attack scenarios, we elaborated on several use cases involving widely used web services. Motivated by the severe consequences of these size-exposing attacks, we proposed an enhanced design for the browser storage, which is likely to be adopted by browser vendors, and discussed a variety of other options that could be employed to prevent adversaries from stealing sensitive information.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This research is partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE). Mathy Vanhoef holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO).

References

- [1] AL FARDAN, N. J., AND PATERSON, K. G. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy* (2013).
- [2] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 75–88.
- [3] BAUER, K., MCCOY, D., GREENSTEIN, B., GRUNWALD, D., AND SICKER, D. Physical layer attacks on unlinkability in wireless lans. In *Privacy Enhancing Technologies* (2009).
- [4] BELSHE, M., PEON, R., AND THOMSON, M. Hypertext transfer protocol version 2 (HTTP/2). RFC 7540, 2015.
- [5] BHARGAVAN, K., LAVAUD, A. D., FOURNET, C., PIRONTI, A., AND STRUB, P. Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Security and Privacy (SP)* (2014).
- [6] BISSIAS, G. D., LIBERATORE, M., JENSEN, D., AND LEVINE, B. N. Privacy vulnerabilities in encrypted HTTP streams. *Lecture notes in computer science* 3856 (2006), 1.
- [7] BORTZ, A., AND BONEH, D. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web* (2007), ACM, pp. 621–628.
- [8] BRIK, V., BANERJEE, S., GRUTESER, M., AND OH, S. Wireless device identification with radiometric signatures. In *Mobile computing and networking* (2008).
- [9] BUGZILLA. Bug 565965 - (doublekey) key cookies on setting domain * toplevel load domain. https://bugzilla.mozilla.org/show_bug.cgi?id=565965, May 2010.
- [10] CAI, X., ZHANG, X. C., JOSHI, B., AND JOHNSON, R. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 605–616.
- [11] CHAPMAN, P., AND EVANS, D. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 263–274.
- [12] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 191–206.
- [13] CHENG, H., AND AVNUR, R. Traffic analysis of SSL encrypted web browsing. *URL citeseer.ist.psu.edu/656522.html* (1998).
- [14] CROSBY, S. A., WALLACH, D. S., AND RIEDI, R. H. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC)* 12, 3 (2009), 17.
- [15] DIERKS, T., AND RESCORLA, E. The transport layer security (TLS) protocol version 1.2. RFC 5246, 2008.
- [16] DUONG, T., AND RIZZO, J. Here come the xor ninjas. In *Ekoparty Security Conference* (2011).
- [17] DYER, K. P., COULL, S. E., RISTENPART, T., AND SHRIMPTON, T. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Security and Privacy (SP)* (2012).
- [18] FAN, Y., LIN, B., JIANG, Y., AND SHEN, X. An efficient privacy-preserving scheme for wireless link layer security. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE* (2008).
- [19] FELTEN, E. W., AND SCHNEIDER, M. A. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and communications security* (2000), ACM, pp. 25–32.
- [20] GELERNTER, N., AND HERZBERG, A. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1394–1405.
- [21] GLUCK, Y., HARRIS, N., AND PRADO, A. BREACH: reviving the CRIME attack. In *Black Hat Briefings* (2013).

- [22] GOOGLE CHROME. Managing HTML5 offline storage. https://developer.chrome.com/apps/offline_storage, February 2016.
- [23] GREENSTEIN, B., MCCOY, D., PANG, J., KOHNO, T., SE-SHAN, S., AND WETHERALL, D. Improving wireless privacy with an identifier-free link layer protocol. In *Mobile systems, applications, and services* (2008).
- [24] GROSSMAN, J. Advanced web attack techniques using GMail. <http://jeremiahgrossman.blogspot.com/2006/01/advanced-web-attack-techniques-using.html>, 2006.
- [25] HINTZ, A. Fingerprinting websites using traffic analysis. In *Privacy Enhancing Technologies* (2003), Springer, pp. 171–178.
- [26] HOMAKOV, E. Using Content-Security-Policy for evil. <http://homakov.blogspot.com/2014/01/using-content-security-policy-for-evil.html>, January 2014.
- [27] ICSI. The ICSI certificate notary. Retrieved 23 Jan. 2016, from <http://notary.icsi.berkeley.edu>.
- [28] IEEE STD 802.11-2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 2012.
- [29] JAGATIC, T. N., JOHNSON, N. A., JAKOBSSON, M., AND MENCZER, F. Social phishing. *Communications of the ACM* 50, 10 (2007), 94–100.
- [30] JAKOBSEN, J. B., AND ORLANDI, C. *A practical cryptanalysis of the Telegram messaging protocol*. PhD thesis, Master Thesis, Aarhus University (Available on request), 2015.
- [31] KELSEY, J. Compression and information leakage of plaintext. In *Fast Software Encryption* (2002), Springer, pp. 263–276.
- [32] KITAMURA, E. Working with quota on mobile browsers. <http://www.html5rocks.com/en/tutorials/offline/quota-research/>, January 2014.
- [33] LANDAU, P. Deanonymizing Facebook users by CSP brute-forcing. <http://www.myseosolution.de/deanonymizing-facebook-users-by-csp-bruteforcing/>, August 2014.
- [34] LEE, S., KIM, H., AND KIM, J. Identifying cross-origin resource status using application cache. In *NDSS* (2015).
- [35] LEKIES, S., STOCK, B., WENTZEL, M., AND JOHNS, M. The unexpected dangers of dynamic JavaScript. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 723–735.
- [36] LUO, X., ZHOU, P., CHAN, E. W., LEE, W., CHANG, R. K., AND PERDISCI, R. HTTPS: Sealing information leaks with browser-side obfuscation of encrypted flows. In *NDSS* (2011).
- [37] MARLINSPIKE, M. New tricks for defeating SSL in practice. *BlackHat DC*, February (2009).
- [38] MATHER, L., AND OSWALD, E. Pinpointing side-channel information leaks in web applications. *Journal of Cryptographic Engineering* 2, 3 (2012), 161–177.
- [39] MICROSOFT. Platform status. <https://dev.windows.com/en-us/microsoft-edge/platform/status/fetchapi>, February 2016.
- [40] MILLER, B., HUANG, L., JOSEPH, A. D., AND TYGAR, J. D. I know why you went to the clinic: Risks and realization of HTTPS traffic analysis. In *Privacy Enhancing Technologies* (2014), Springer, pp. 143–163.
- [41] MOORE, T., AND EDELMAN, B. Measuring the perpetrators and funders of typosquatting. In *Financial Cryptography and Data Security*. Springer, 2010, pp. 175–191.
- [42] MOZILLA DEVELOPER NETWORK. Browser storage limits and eviction criteria. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Storage_limits_and_eviction_criteria, October 2015.
- [43] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIJSESENS, F., AND VIGNA, G. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 736–747.
- [44] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox: Practical cache attacks in JavaScript. *arXiv preprint arXiv:1502.07373* (2015).
- [45] PEON, R., AND RUELLAN, H. HPACK: Header compression for HTTP/2. RFC 7541, 2015.
- [46] RANGANATHAN, A., AND SICKING, J. File API. *W3C Working Draft* (2012).
- [47] RESCORLAN, E. HTTP over TLS. RFC 2818, 2000.
- [48] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 199–212.
- [49] RIZZO, J., AND DUONG, T. The CRIME attack. In *EKOparty Security Conference* (2012), vol. 2012.
- [50] ROESNER, F., KOHNO, T., AND WETHERALL, D. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 12–12.
- [51] RYDSTEDT, G., BURSZTEIN, E., BONEH, D., AND JACKSON, C. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web 2* (2010), 6.
- [52] SCHINZEL, S. An efficient mitigation method for timing side channels on the web. In *2nd International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)* (2011).
- [53] SEGALL, L. An app called Telegram is the 'hot new thing among jihadis'. <http://money.cnn.com/2015/11/17/technology/isis-telegram/>, November 2015.
- [54] SOOD, A. K., AND ENBODY, R. J. Malvertising: Exploiting web advertising. *Computer Fraud & Security* 2011, 4 (2011), 11–16.
- [55] SSL PULSE. Survey of the SSL implementation of the most popular web sites. <https://www.trustworthyinternet.org/ssl-pulse/>, February 2016.
- [56] STATCOUNTER. GlobalStats. <http://gs.statcounter.com/#all-browser-ww-monthly-201501-201601>, January 2016.
- [57] SUN, Q., SIMON, D. R., WANG, Y.-M., RUSSELL, W., PADMANABHAN, V. N., AND QIU, L. Statistical identification of encrypted web browsing traffic. In *Security and Privacy* (2002).
- [58] TERADA, T. Identifier based XSS attacks. <https://www.mbsd.jp/Whitepaper/xssi.pdf>, March 2015.
- [59] TOR. Isolate HTTP cookies according to first and third party domain contexts. <https://trac.torproject.org/projects/tor/ticket/3246>, May 2011.
- [60] VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1382–1393.
- [61] VANHOEF, M., AND PIJSESENS, F. Advanced Wi-Fi attacks using commodity hardware. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACM, pp. 256–265.

- [62] VANHOEF, M., AND PIJSSSENS, F. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security Symposium* (2015).
- [63] W3C. Offline web applications. <https://www.w3.org/TR/offline-webapps/>, May 2008.
- [64] W3C. Same-origin policy. https://www.w3.org/Security/wiki/Same_Origin_Policy, January 2010.
- [65] W3C. Quota management API. <https://www.w3.org/TR/quota-api/>, December 2015.
- [66] W3C. Service Workers. <https://www.w3.org/TR/service-workers/>, June 2015.
- [67] WAGNER, D., SCHNEIER, B., ET AL. Analysis of the SSL 3.0 protocol. In *The Second USENIX Workshop on Electronic Commerce Proceedings* (1996), pp. 29–40.
- [68] WANG, T., AND GOLDBERG, I. Comparing website fingerprinting attacks and defenses. Tech. rep., Technical Report 2013-30, CACR, 2013. <http://cacr.uwaterloo.ca/techreports/2013/cacr2013-30.pdf>, 2014.
- [69] WEBKIT. Implement fetch API. https://bugs.webkit.org/show_bug.cgi?id=151937, December 2015.
- [70] WHATWG. Storage. <https://storage.spec.whatwg.org/>, August 2015.
- [71] WiGLE. WiFi encryption over time. Retrieved 6 Feb. 2016 from <https://wgle.net/enc-large.html>.
- [72] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 719–732.
- [73] ZALEWSKI, M. *The tangled Web: A guide to securing modern web applications*. No Starch Press, 2012.
- [74] ZHANG, F., HE, W., CHEN, Y., LI, Z., WANG, X., CHEN, S., AND LIU, X. Thwarting Wi-Fi side-channel analysis through traffic demultiplexing. *Wireless Communications, IEEE Transactions on* 13, 1 (2014), 86–98.
- [75] ZHANG, F., HE, W., AND LIU, X. Defending against traffic analysis in wireless networks through traffic reshaping. In *Distributed Computing Systems (ICDCS)* (2011).
- [76] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 990–1003.
- [77] ZHOU, X., DEMETRIOU, S., HE, D., NAVEED, M., PAN, X., WANG, X., GUNTER, C. A., AND NAHRSTEDT, K. Identity, location, disease and more: Inferring your secrets from Android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1017–1028.

Trusted browsers for uncertain times

David Kohlbrenner*
UC San Diego

Hovav Shacham†
UC San Diego

Abstract

JavaScript in one origin can use timing channels in browsers to learn sensitive information about a user’s interaction with other origins, violating the browser’s compartmentalization guarantees. Browser vendors have attempted to close timing channels by trying to rewrite sensitive code to run in constant time and by reducing the resolution of reference clocks.

We argue that these ad-hoc efforts are unlikely to succeed. We show techniques that increase the effective resolution of degraded clocks by two orders of magnitude, and we present and evaluate multiple, new implicit clocks: techniques by which JavaScript can time events without consulting an explicit clock at all.

We show how “fuzzy time” ideas in the trusted operating systems literature can be adapted to building trusted browsers, degrading all clocks and reducing the bandwidth of all timing channels. We describe the design of a next-generation browser, called Fermata, in which all timing sources are completely mediated. As a proof of feasibility, we present Fuzzyfox, a fork of the Firefox browser that implements many of the Fermata principles within the constraints of today’s browser architecture. We show that Fuzzyfox achieves sufficient compatibility and performance today by privacy-sensitive users.

In summary:

- We show how an attacker can measure durations in web browsers without querying an explicit clock.
- We show how the concepts of “fuzzy time” can apply to web browsers to mitigate all clocks.
- We present a prototype demonstrating the impact of some of these concepts.

1 Introduction

Web browsers download and run JavaScript code from sites a user visits as well as third-party sites like ad networks, granting that code access to system resources through the DOM. Keeping that untrusted code from taking control of the user’s system is the *confinement* problem. In addition, browsers must ensure that code running in one origin does not learn sensitive information

about the user’s interaction with another origin. This is the *compartmentalization* problem.

A failure of confinement can lead to a failure of compartmentalization. But JavaScript can also learn sensitive information without escaping from its sandbox, in particular by exploiting *timing side channels*. A timing channel is made possible when an attacker can compare a *modulated clock*—one in which ticks arrive faster or slower depending on a secret—to a *reference clock*—one in which ticks arrive at a consistent rate. For example, browsers allow web pages to apply SVG transformations to page elements, including cross-origin frames, via CSS. Paul Stone showed that a fast-path optimization in the `feMorphology` filter created a timing attack that allowed attackers to steal pixels or sniff a user’s browsing history, using `Window.requestAnimationFrame()` as a modulated clock [24]. More recently, Oren et al. showed that, in the presence of a high-resolution reference clock like `performance.now`, attackers could use JavaScript `TypedArrays` to measure instantaneous load on the last-level processor cache [19].

Browser vendors are aware of the danger that timing channels pose compartmentalization and have made efforts to address it.

First, they have attempted to eliminate modulated clocks by making any code that manipulates secret values run in constant time. In a hundred-message Bugzilla thread, for example, Mozilla engineers decided to address Stone’s pixel-stealing work by rewriting the `feMorphology` filter implementation using constant-time comparisons.¹

Second, they have attempted to reduce the resolution of reference clocks available to JavaScript code. In May, 2015, the Tor Browser developers reduced the resolution of the `performance.now` high-resolution timer to 100 ms as an anti-fingerprinting measure.² In late 2015, some major browsers (Chrome, Firefox) applied similar patches (see Figure 1), reducing timer resolution to 5 µs to defeat Oren et al.’s cache timing attack [19].

These efforts are unlikely to succeed, because they seriously underestimate the complexity of the problem.

First, eliminating every potential modulated clock would require an audit of the entire code base, an ambitious undertaking even for a much smaller, simpler system such as a microkernel [3]. Indeed, the Mozilla fix for `feMorphology` did not consider the possibility that

*dkohlbre@cs.ucsd.edu
†hovav@cs.ucsd.edu

floating-point instructions execute faster or slower depending on their inputs, allowing pixel-stealing attacks even in supposedly “constant-time” code [1].

Second, there are many ways by which JavaScript code might synthesize a reference clock besides naively querying `performance.now`. In this paper, we show that *clock-edge detection* allows JavaScript to increase the effective resolution of a degraded `performance.now` clock by two orders of magnitude. We also present and evaluate multiple, new *implicit clocks*: techniques by which JavaScript can time events without consulting an explicit clock like `performance.now` at all. For example, videos in an HTML5 `<video>` tag are decoded in a separate thread. JavaScript can play a simple video that changes color with each frame and examine the current frame by rendering it to a canvas. This immediately gives an implicit clock with resolution 60Hz, and the resolution can be improved using our techniques.

In short, timing channels pose a serious danger to compartmentalization in browsers; browser vendors are aware of the problem and are attempting to address it by eliminating or degrading clocks attackers would rely on, but their ad-hoc efforts are unlikely to succeed. Our thesis in this paper is that the problem of timing channels in modern browsers is analogous to the problem of timing channels in trusted operating systems and that ideas from the trusted systems literature can inform effective browser defenses. Indeed, our description of timing channels as the comparison of a reference clock and a modulated clock is due to Wray [28], and our fuzzy mitigation strategy technique is directly inspired by Hu [10]—both papers resulting from the VAX VMM Security Kernel project, which targeted an A1 rating [12].

In this paper, we show that “fuzzy time” ideas due to Hu [10] can be adapted to building trusted browsers. Fuzzy time degrades *all* clocks, whether implicit or explicit, and it reduces the bandwidth of all timing channels. We describe the properties needed in a trusted browser where all timing sources are completely mediated. Today’s browsers tightly couple the JavaScript engine and the DOM and would need extensive redesign to completely mediate all timing sources. As a proof of feasibility, we present Fuzzyfox, a fork of the Firefox browser that works within the constraints of today’s browser architecture to degrade timing sources using fuzzy time. Fuzzyfox demonstrates a principled clock fuzzing scheme that can be applied to both mainstream browsers and Tor Browser using the same mechanics. We evaluate the performance overhead and compatibility of Fuzzyfox, showing that all of its ideas are suitable for deployment in products like Tor Browser and a milder version are suitable for Firefox.

```
double PerformanceBase::clampTimeResolution
(double timeSeconds)
{
    const double resolutionSeconds =
        0.000005;
    return floor(timeSeconds /
        resolutionSeconds) *
        resolutionSeconds;
}
```

Figure 1: Google Chrome `performance.now` rounding code

```
// Find minor ticks until major edge
function nextedge(){
    start = performance.now();
    stop = start;
    count = 0;

    while(start == stop){
        stop = performance.now();
        count++;
    }

    return [count,start,stop];
}

// run learning
nextedge();
[exp,pre,start] = nextedge();

// Run target function
attack();

// Find the next major edge
[remain,stop,post] = nextedge();

// Calculate the duration
duration = (stop-start)+((exp-remain)/exp)*
grain;
```

Figure 2: Clock-edge fine-grained timing attack in JavaScript

2 Clock-edge attack

Web browser vendors have attempted to mitigate timing side channel attacks like [19] by rounding down the explicit clocks available to JavaScript to some grain g . For example, Google Chrome and Firefox have implemented a $5\mu s$ grain. Figure 1 shows the C++ code used for rounding a `performance.now` call in Google Chrome. Tor Browser makes a different privacy and performance tradeoff and has implemented an aggressive 100ms grain.

Unfortunately, rounding down does not guarantee that an attacker cannot accurately measure timing differences smaller than g . We present the *clock-edge* technique for improving the granularity of time measurements in the context of JavaScript clocks. Experi-

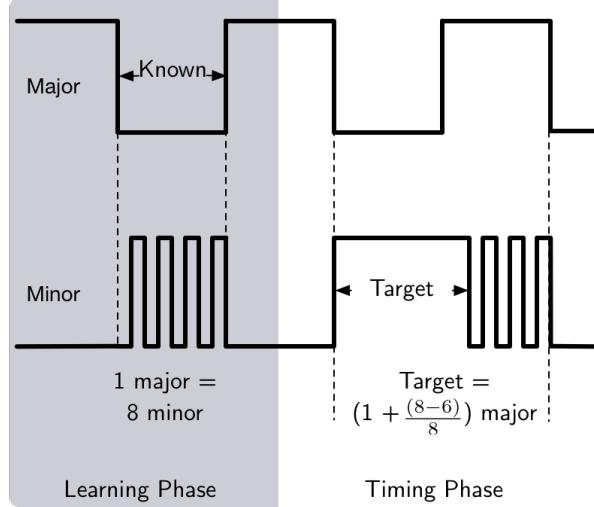


Figure 3: Clock-edge learning and timing

tally, this technique results in an increase in resolution of at least two orders of magnitude to large grained clocks. This technique can be generalized to any pair of clocks: a *major* clock, which has a known large period, and a *minor* clock, which has a short unknown period. The major clock is used to establish the period of the minor clock, and together they can time events with more accuracy than alone.

Consider the case of a page wishing to time some JavaScript function `attack()` with a granularity smaller than some known performance.now grain g . The major clock in this case is the degraded performance.now, and we use a tight incrementing `for` loop as the minor clock. Figures 2 and 3 show how a page might execute this technique and a visual representation of the process.

The page first learns the average number of loop iterations (L_{exp}) between the major clock ticks C_{l1} and C_{l2} . After learning, the page then runs until a major clock edge is detected (C_{start}) and then executes `attack()`. When `attack()` returns at major clock time C_{stop} , the page runs the minor clock (for L_{remain} ticks) until the next major clock edge (C_{post}) is detected. The page then calculates the duration of `attack()` as $(C_{stop} - C_{start}) + g * (L_{exp} - L_{remain}) / (L_{exp})$. In the case of g not remaining constant, we scale the L_{exp} by $(C_{post} - C_{stop}) / (C_{l2} - C_{l1})$ and set $g = C_{post} - C_{stop}$.

Since $(L_{exp} - L_{remain}) / (L_{exp})$ represents a fractional portion of g , the duration measurement can plausibly obtain measurements as fine grained as g / L_{exp} . Thus, as long as the attacker has access to a suitable minor clock, the degradation of a major clock to g by rounding does not ensure an attacker cannot measure at a grain less than g .

Grain(ms)	Minor	Measured Durations(ms)			
		0.003	0.030	0.298	3.033
None	-	0.002	0.029	0.299	3.103
0.001	2	0.004	0.032	0.304	3.031
0.005	94	0.003	0.030	0.298	2.998
0.01	192	0.003	0.030	0.303	3.009
0.08	1649	0.011	0.027	0.299	3.006
0.1	1965	0.053	0.038	0.296	3.010
1	20470	0.112	0.208	0.332	3.159
10	193151	0.436	0.469	0.560	3.330
100	1928283	1.045	1.076	1.294	3.437
500	9647265				

Table 1: Results for running the clock-edge fine-grained timing attack against various grain settings. Averages for 100 runs shown.

Table 1 shows the results of applying the clock-edge technique on a degraded performance.now major clock on 4 different targets at different grains. The code in figure 2 is an abbreviated version of the testing code. Each duration column represents a different number of iterations in the `attack()` function, which is an empty `for` loop. The minor ticks column indicates the number of iterations the learning phase detected that each major tick takes. The “None” row indicates the runtime of attack with no rounding enabled, and other rows indicate the durations measured at different grain settings using the clock-edge technique. Measurements were performed with a modified build of Firefox that enabled setting arbitrary grains via JavaScript.

As table 1 shows, the clock-edge attack recovers durations significantly smaller than the grain settings. Notably, grains in the millisecond and higher range still permit the differentiation of events lasting only tens of μ s!

Simply rounding down the available explicit clocks only has a notable impact if the attacker is attempting to differentiate between events each lasting less than a microsecond, at which level the clock-edge attack often provides no additional resolution to the rounded clock.

3 Measuring time in browsers without explicit clocks

In this section, we demonstrate different methods an attacker can use to measure the duration of events in JavaScript. An attacker wishing to mount a timing attack against a web browser is not restricted to the use of performance.now for timing measurements, this section will present a number of alternative methods available. Browser features that enable these measurements are *implicit clocks*. Depending on the how the target and the clock interact with the JavaScript runtime, we define them as *existing* or *exitless*. We do not present an exhaustive list of implicit clocks. Rather, this section

should be considered the tip of the iceberg for clock techniques in browsers.

3.1 Measurement targets

Recall that the adversary’s goal in a timing attack is to measure the duration of some event and differentiate between two or more possible executions. We assume our adversary’s goal is to measure the duration of some piece of JavaScript `target()` or to measure the time until some event `target` fires a callback. There are many potential targets, exemplified by two different timing attacks on web browsers. We categorize targets and attacks into exiting and exitless and describe a canonical example for each.

3.1.1 Exiting targets: privacy breaches with `requestAnimationFrame`

Previous work [1] [24] has shown several different ways to achieve history sniffing or cross frame pixel reading via timing the rendering of an SVG filter over secret data. Andryscy et al [1] demonstrate a timing attack on privacy that differentiates pixels based on how long rendering an SVG convolution filter takes. This timing requires that the attacking JavaScript know exactly when the SVG filter is applied to the target and when the SVG filter finishes rendering. This is accomplished by sampling a high resolution time stamp (`performance.now`) when applying the CSS style containing the filter and when a callback for `requestAnimationFrame` fires. In this case, JavaScript must exit to allow some other computation to occur and then receives a notification via a callback that the event has completed. We refer to this type of target as an *exiting* target, as it exits the JavaScript runtime before completion.

3.1.2 Exitless targets: cache timing attacks from JavaScript

Conversely, there are *exitless* targets, such as Oren et al’s [19] cache timing attack. This attack does not need to exit JavaScript for the `target` to run, instead they need only perform some synchronous JavaScript function call, and measure the duration of it. Any exitless target can be scheduled in callbacks, thus making it an exiting target, but an exiting target cannot be run in an exitless manner.

3.2 Implicit clocks in browsers

Supposing that all explicit clocks were removed from the browser, it is still possible that a motivated attacker can measure fine-grained durations. Rather than query an explicit clock, the attacker can find some other feature of the browser that has a known or definable execution time and use that as an *implicit clock*.

We did not test any clocks that resolve durations at an external observer, such as a cooperating server. For ex-

Description	Clock type		
	Firefox	Chrome	Safari
Explicit clocks	L	L	L
Video frames	L	L	L
Video played	X	L	L
WebSpeech API	L	+	—
<code>setTimeout</code>	X	X	X
CSS Animations	X	X	X
WebVTT API	X	X	X
Rate-limited server	X	X	X

Table 2: Implicit clock type in different browsers
L Exitless , X Exiting , — Not implemented, + Buggy

ample, a piece of JavaScript could generate a network request, run a `target`, and then generate another network request. These clocks are mitigated by the defenses discussed in section 4.

We observe that just as with exiting and exitless targets, there are exiting and exitless implicit clocks. We will refer to a clock or timing method that does not need to leave JavaScript execution for the value reported by the clock to change as *exitless*. Similarly, a timing method that requires JavaScript execution to exit before time moves forward is *exiting*.

All exitless clocks can work for both exiting and exitless targets. However, an exitless target cannot function with an exiting clock, as the execution of the target will take control of the main thread, stopping regular callbacks or events that the exiting clock needs from firing. There may be exotic exiting clocks that do not have this restriction, but all of the ones detailed below do. An exitless attack requires using both an exitless target and clock (such as in the cache timing attack.)

Depending on the implementation of a browser feature, the clock technique may be exiting or exitless. A good example is the updating of the played information for an `<audio>` or `<video>` tag. This information is updated asynchronously to the main browser thread in Google Chrome but will not update during JavaScript execution in Firefox. Thus, it can be used to construct a exitless clock in Chrome but only an exiting clock in Firefox.

See table 2 for how the following clocks manifest in Chrome 48 (stable), Firefox³, and Safari 9.0.3.

3.2.1 Exitless clocks

Since JavaScript is single threaded and non-preemptable, exitless clocks do not have to worry about the scheduling of other JavaScript callbacks or any other events occurring between the target and timing measurements. By the semantics of JavaScript, an exitless clock is considered a run-to-completion violation[18] and is a bug. Any time JavaScript can observe changes caused externally during

a single callback qualifies as such a bug; it is only when their timing is dependable that we can construct a clock. Mozilla has explicitly stated their goal to make SpiderMonkey (the Firefox JavaScript engine) free of run-to-completion violations.

We found several exitless clocks available to JavaScript in different browsers.

1. Explicit clock queries. While expected, explicit clock queries are run-to-completion violations and expose the most accurate timing data. `performance.now` is the best source of explicit timing data in JavaScript.
2. Video frame data. By rendering a `<video>` to `<canvas>`, JavaScript can recover the current video frame. Since the video updates asynchronous to the browser event loop, this can be used to get a fine grained time-since-video-start value repeatedly.

On Firefox, video frame data updates at 60 FPS, giving a granularity of 17ms. We can load a video at 120FPS, which does not allow JavaScript access to new frames faster, but the frames JavaScript gets are a more accurate clock. We demonstrate this by generating a long-running video at 120FPS that changes the color of the entire video every frame. Thus, by sampling the current color via rendering the video to `<canvas>`, the page can measure how much time has elapsed since the video started. Video can be rendered off-screen or otherwise invisible to the user and will still update at 60FPS, making it an ideal choice for an implicit clock. We have also found that using multiple videos and averaging the reported time between them provides additional accuracy.

3. WebSpeech API. This can start/stop the speaking of a phrase from JavaScript and will give a high-resolution duration measurement when stopped. The WebSpeech API allows JavaScript to define a `SpeechSynthesisUtterance`, which contains a phrase to speak. This process can be started with `speak()` and then stopped at any time with `cancel()`. The cancellation can fire a callback whose event contains a high resolution duration of how long the system was speaking for. Thus, the attacker can start a phrase, run some target JavaScript function, and then cancel the phrase to obtain a timing target. Note that while the callback must fire to get the duration value, the duration measurement stops when `window.speechSynthesis.cancel()` is called, not when the callback eventually fires. This makes the WebSpeech API a pseudo-exitless clock in Firefox, even though we must technically wait for a callback to get back the duration measurement. Time moved forward, we just couldn't observe repeatedly. Since we can only measure the clock by stopping it,

the clock-edge technique cannot be used to enhance the accuracy of the clock.

The WebSpeech API is only supported in Firefox 44+, and on many systems will need to be manually enabled in `about:config`. Additionally, unless the OS has speech synthesis support, the clock cannot be used as it will never start speaking. Ubuntu can get this support by installing the `speech-dispatcher` package.

4. SharedArrayBuffers. While we did not test these, as the implementation is still ongoing, any sort of shared memory between JavaScript instances constitutes an exitless clock. As demonstrated in [23], this can be used as a very precise clock in real attacks.

3.2.2 Exiting clocks

Exiting clocks are far more numerous but also significantly less useful to an attacker, as their measurements and target execution are unlikely to be continuous.

1. `setTimeout`. Set to fire every millisecond, these then set a globally visible “time” variable when they do. This is the most basic of the exiting clocks. We set timeouts every millisecond as this is lowest resolution that can be set.
2. CSS animations. Set to finish every millisecond, these then set a globally visible “time” variable in their completion callback. These behave almost identically to `setTimeouts` and are measured in the same way.
3. WebVTT. This API can set subtitles for a `<video>` with up to millisecond precision and check which subtitles are currently displayed. The WebVTT interface provides a way for `<video>` elements to have subtitles or captions with the `<track>` element. These captions are loaded from a specified VTT file, which can specify arbitrary subtitles to appear for unlimited duration with up to millisecond precision. By setting a different subtitle to appear every millisecond, the page can determine how much time has elapsed since the video started by checking the `track.activeCues` attribute of the `<track>` element. This only updates when JavaScript is not executing.
4. A rate limited download. Using a cooperating server to send a file to the page at a known rate causes regular progress updates to be queued in callbacks. Using the `onprogress` event for XMLHttpRequests (XHRs), the page can get a consistent stream of callbacks to a clock update function. Note that the rate of these callbacks is related to the size of the file being retrieved, as well as the upload rate of the server.

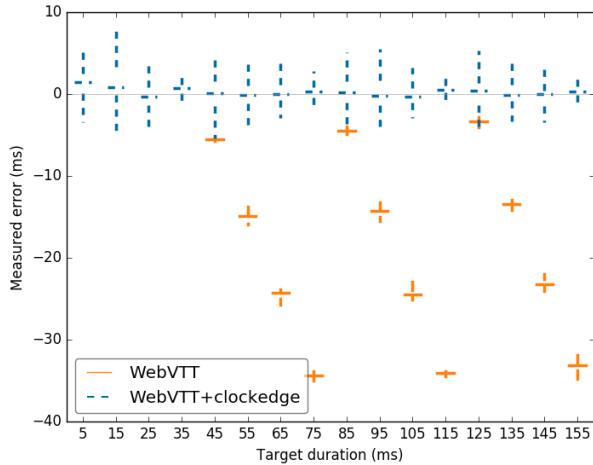


Figure 4: WebVTT error measurements with and without clock-edge technique

In our experiments, we used a file 100mB in size, with a server rate limited to 100kB/s using the Linux utility `trickle`. The page then assumes that the server is sending data at exactly 100kB/s and has an initial learning period to determine the rate at which the `onprogress` callbacks fire. After that is complete, the page can continue running as usual, with the assumption that it now has a regular callback firing at the calculated rate. Note that the `onprogress` events can also be requested to fire during the loading of `<video>` elements.

5. Video/audio tag played data. These contain the intervals of the media object that have thus far been played. By checking the furthest played point repeatedly, we can measure the duration of events. In Firefox, this only updates after JavaScript exits, but in Chrome, it updates asynchronously (making it an exitless clock for Chrome).
6. Cooperating iframes/popups from same origin. By creating a popup in the same origin, or by embedding iframes from the origin, two pages can cooperate and act on the same DOM elements. In our testing there was no way to get exitless DOM element manipulations updates in this situation. Thus, this case reduces to the `setTimeout` case or another similar method. We do not present any timing results for these clocks. Critically, if a method of sharing DOM element updates exitlessly were found this would become an exitless clock.

3.3 Performance of implicit clocks

The granularity, precision, and accuracy of implicit clocks varies widely by technique. We observe that

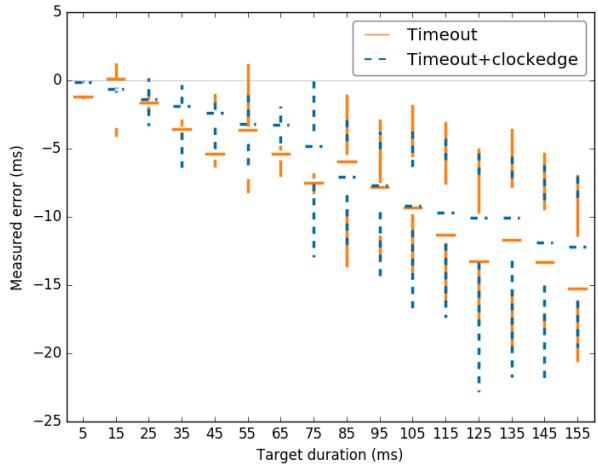


Figure 5: `setTimeout` error measurements with and without clock-edge technique

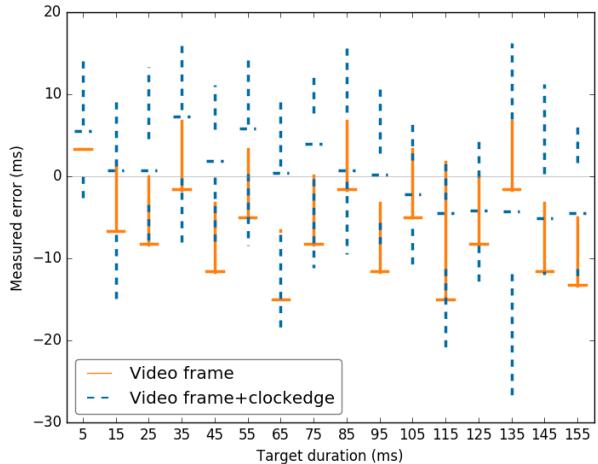


Figure 6: Video frame error measurements with and without clock-edge technique

most implicit clocks can be improved with the clock-edge technique from section 2. By substituting the `performance.now` major clock with the implicit clock technique, and using a suitable minor clock, most techniques showed notable improvements in accuracy. In this case, we want to examine how easy it would be to differentiate two different duration events. Thus, tight error bounds that are consistent are ideal.

Applying the clock-edge technique to exitless clocks only requires the replacement of the explicit `performance.now` call to some other exitless clock; no change to the minor clock is needed. Existing clocks require a new minor clock technique; instead of a tight loop, the minor clock must schedule regular timeouts that check the state of the implicit major clock. Otherwise, the exiting major clock would not change

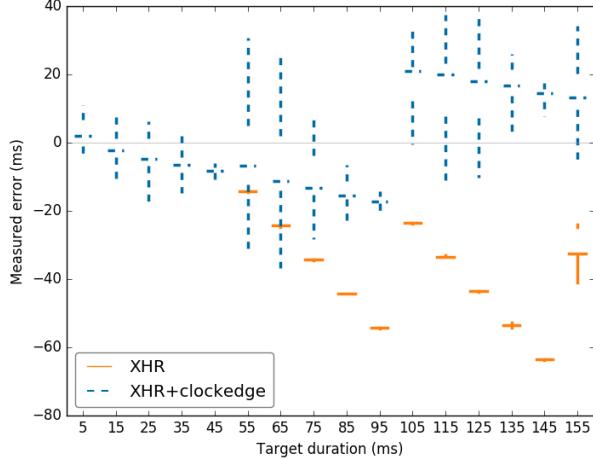


Figure 7: Throttled XMLHttpRequest error measurements with and without clock-edge technique

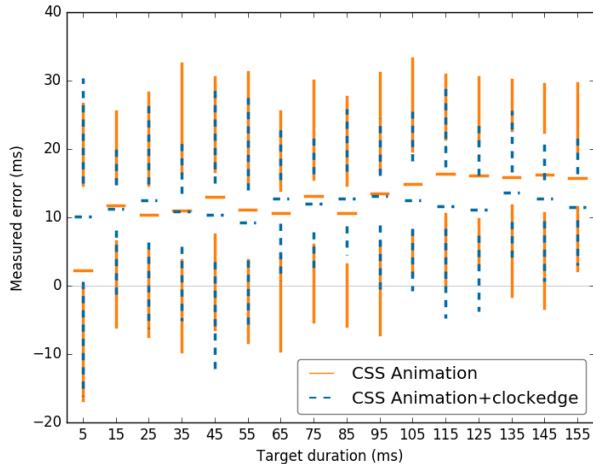


Figure 8: CSS animation error measurements with and without clock-edge technique

state while the minor clock is running. While repeated `setTimeout` calls would work, `setTimeout` of 0 is actually a 4ms timeout per the HTML5 spec, making it a major clock. Instead, we use repeated `postMessage` calls to the current window. These execute at a much higher rate, but the period is unknown. Thus the new implicit major clock now has a fast, unknown period minor clock, just as in the exitless case.

Measurements were done with the same Firefox as in section 2. Error (y values) was calculated as the difference between the clock technique measurement and the actual duration as reported by `performance.now`. Target durations (x values) are the expected duration (N milliseconds) of the target event, which may differ slightly from actual duration due to system load or even the implicit clocks themselves interfering in the case of

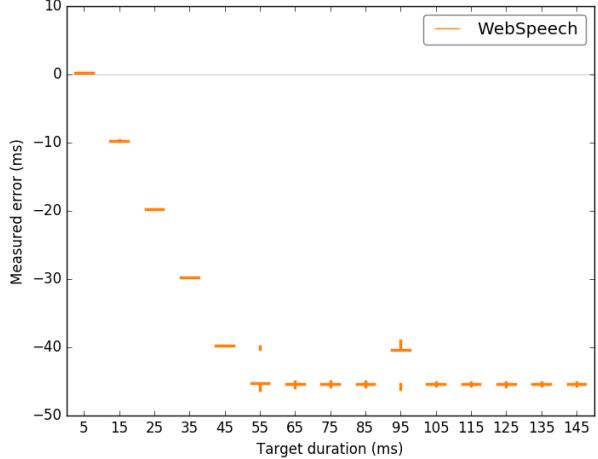


Figure 9: WebSpeech error measurements without clock-edge technique

existing clocks. Each target was measured 100 times, with measured durations of 0 or less removed. While actual durations varied slightly from expected, there was not considerable noise.

The exitless target we measure is a loop that runs for N milliseconds, as determined by `performance.now`. Our exiting target is a `setTimeout` for N milliseconds.

Figures 4, 5, 6, 7, 8, and 9 show the clock technique error with and without clock-edge improvements for a variety of clock techniques described above. WebSpeech has no clockedge data for the reasons detailed in 3.2.1. Note that the y-axis differs per figure, to allow for easier comparison between clock-edge and non-clock-edge results. As can be seen in WebVTT, throttled XHRs, and video frame data, many clock techniques have a large native period that they operate at. These large periods leave plenty of space for clock-edge to improve accuracy. WebVTT shows massive improvement in the clock-edge case due to the *precision* of its major clock ticks; the more precise the original technique, the more accurate clock-edge can be.

Figures 11 and 10 show the comparison of the averaged error for all techniques and all techniques with clock-edge respectively. The closer a line is to 0 on these graphs, the more accurate the averaged measurements will be for that technique. Again, the exceptional accuracy of WebVTT with clock-edge for long-duration events is evident.

4 Fermata

In this section we describe *Fermata*, a theoretical browser design that provably degrades all attacker visible clocks. Sections 5 and 6 describe our prototype implementation, Fuzzyfox, and an evaluation. Fermata is

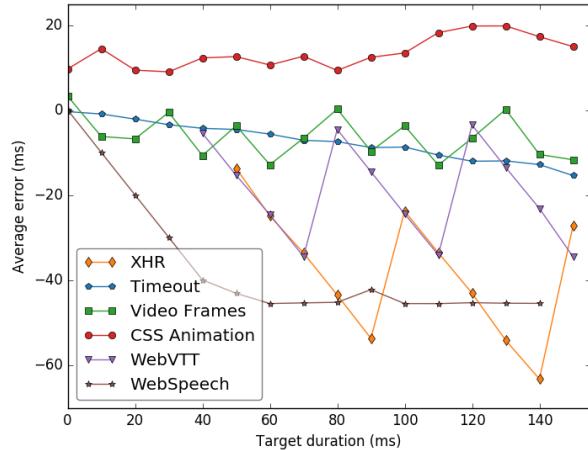


Figure 10: Average error for all clock techniques without clock-edge

an adaptation of the *fuzzy time* operating systems concept detailed in [10] to web browsers.

Since browser vendors have expressed an interest in degrading time sources available to JavaScript, we present Fermata as a design ideal for a browser that will provably degrade all clocks. Fermata’s goal is to provide the attacker with only time sources that update at a rate such that all possible timing side channels have a bounded maximum bandwidth. This includes the use of all the implicit clocks described in section 3 as well as any other such clock unknown to us.

4.1 Why Fermata?

We propose Fermata because we believe that attempting to audit and secure all possible channels in a modern web browser is infeasible. The evaluation of a provable security focused microkernel found several tricky timing channels [3]. In that case, the microkernel was designed to be audited and already had a number of concerns accounted for; this is not true in the case of a modern web browser. Rather than allow any unknown channel to leak data arbitrarily until fixed, Fermata restricts all known and unknown channels to leak at or below a target acceptable rate.

Fermata proposes a principled alternative to the “find and mitigate all clocks” methodology that Tor Browser has already begun. Rather than manually examine every DOM manipulation, extension, or new feature, Fermata requires minimal defined interfaces between all components. By automatically proving that all information passes through these interfaces and that all such interfaces are subject to the fuzzing process, Fermata will drastically reduce the burden of code that needs to be examined. This is analogous to other such approaches in

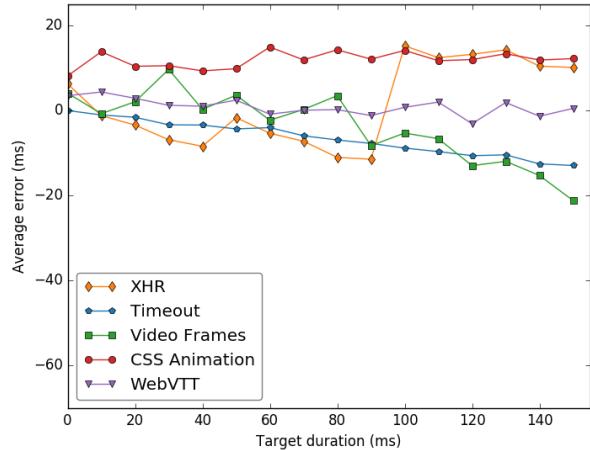


Figure 11: Average error for all clock techniques with clock-edge where available

the programming languages and formal software community.

Limiting the channel bandwidth for an attacker leaking information is not a complete solution to timing attacks on browsers, but it is a realistic one. Previous attacks on history sniffing [1] [24] have consistently cropped up. These privacy breaches are only as valuable as the amount of data they can collect. Learning that a user has visited 2-3 websites is not likely to create a unique profile of them. Learning tens of thousands of websites likely would [27]. History sniffing attacks are therefore classified based on how fast they can extract the visited status of a URL. By limiting the rate at which this information can leak, Fermata can make history sniffing impractical. As an example, [27] indicates that an attacker may need to sniff in excess of 10,000 URLs to create a reasonable fingerprint for a user. With an attack like [24] the attacker can read 60 or more URLs per second. Previous attacks not utilizing timing side channels read in excess of 30,000 URLs per second.

We expect that Fermata would allow a channel bandwidth of ≤ 50 bits per second in the general case, and ≤ 10 for security critical workflows. The protection is even stronger than initially obvious, as attacks that rely on small timing differences are entirely unusable. Only attacks that can scale their detection thresholds up (for example, Andryscy et al [1]) can still leak data. If the attack relies on a small, inherent microarchitecture timing, such as Oren et al’s [19] cache timing attack, which measured differences around 100ns, this timing difference may no longer be perceptible at all. An additional benefit is that many of these attacks require intensive learning phases, during which many measurements must be taken to establish timing profiles. Fermata would force this learning phase to take significantly longer, adding

to the time-per-bit of information extracted. From this survey of previous attacks, we believe that a strong limitation on channel bandwidth represents a powerful defense against timing attacks in browsers.

4.2 Threat model

We define our attacker as the canonical web attacker who legitimately controls some domain and server. They are able to cause the victim to visit this page in Fermata and run associated JavaScript. The attacker thus has two viewpoints we must consider: any external server controlled by the attacker and the JavaScript running in Fermata.

The attacker in our case possesses a timing side-channel vulnerability they wish to use on Fermata. The specific form of the vulnerability does not matter, only that it can be abstracted as a single JavaScript function that is called either synchronously or asynchronously. The attacker uses the duration of this function to derive secret information about the victim, possibly repeatedly.

We do not present a solution for plugins like Adobe Flash or Java applets. Significant changes to the runtime of these plugins on-par with Fermata itself would need to be made for them to be similarly resistant. Considering the number of known vulnerabilities and privacy disclosures in most of these plugins, we do not believe they should be a part of a browser design focusing on security and privacy. Alternatively, such plugins should be disabled during sensitive work flows.

The attacker succeeds against Fermata if they are able to extract bits using their side channel at a higher rate than the maximum channel bandwidth.

4.3 Design goals and challenges for Fermata

Fermata must mediate the execution of JavaScript to remove all exitless clocks and degrade all exiting clocks. This would include mediating and randomly delaying all network I/O, local I/O, communication between JavaScript instances (iframes, workers, etc), and communication to other processes (IPC). If Fermata were additionally able to make all DOM accesses by JavaScript asynchronous and delay them in the same principled fashion, this would accomplish our goals. The coupling of JavaScript’s globally accessible variables to the DOM represents the most significant challenge to such a design and presents a shared state problem not found in the model for this work [10].

Given this shared state problem, Fermata has two options for JavaScript: redesign JavaScript execution to be entirely asynchronous or degrade explicit clocks and mediate known APIs in a principled manner. The former provides a formal guarantee but cannot be done in current browser architectures. We explore options for the latter later in this section and in Fuzzyfox.

4.4 Fermata guarantees

We believe that the analysis of Hu’s fuzzytime by Gray in [5] applies to Fermata. The means that we can place an upper bound on the leakage rate of Fermata at $\frac{1}{g/2}$ symbols per second, assuming the median tick rate of $\frac{g}{2}$.

As in [5], we assume that increasing the size of the alphabet used will provide negligible benefits. Thus, this bound is an upper bound for the bits-per-second leakage rate of Fermata. We view the vulnerable functionality targeted by the attacker in the strongest possible way: the attacker has complete control over when and how it leaks timing information. This is effectively the high/low privilege *covert channel* scenario the fuzzytime disk contention channel is analyzed under. Similarly, in Fermata, the leaking feature may have access to the same fuzzy clock as the attacker. This allows them to synchronize instantly from “low to high” privilege as in the fuzzytime analysis. Thus, the side channel threat model Fermata operates under is a subset of the fuzzy time model.

There is further analysis of the capacity of covert channels with fuzzy time defenses in [6]. The general case problem of covert channel capacity under fuzzy time appears to be intractable but can be bounded under specific circumstances.

4.4.1 Transmitted bits vs information learned

Fermata makes a guarantee about the actual transmitted bitrate of some side channel. This has obvious benefits in the case of leaking a CSRF token or a cryptographic key: the bits the attacker needs to learn equals the number of bits in the key or token. However, this becomes trickier to quantify with a goal like history sniffing where the details of the side channel can influence what the attacker learns with each leaked bit.

Consider a timing side channel that can indicate if a single URL has been visited by the victim one at a time. Each time the channel is used one bit of information (visit status of the URL) is leaked. If the attacker wishes to learn the visit status of 10,000 URLs they must check each individually.

If instead a timing side channel could indicate if any URLs from an arbitrary set were visited, the attacker could use this along with prior knowledge that almost all URLs have not been visited to learn about more URLs in less bits. Given some set of 10,000 URLs, the side channel indicates that at least one was visited and then, in a divide-and-conquer approach, the first half indicates that none were visited. How many bits were leaked? Two bits were transmitted: that some URLs were visited in the 10,000, and that no URLs in the first 5,000 were visited. However, we have learned the visit status of 5,000 URLs. This is only possible because the attacker can assume the majority of URLs are not visited.

We believe that Fermata’s guarantees still constitute a

valuable defense against using timing side channels for history sniffing. First, not all history sniffing side channels have allowed checking the visit status of batches of URLs. In these cases Fermata limits learning the visit status of each URL individually. Second, if the attacker wishes to learn specific URLs from the browsing history (ex: to launch a targeted phishing attack), rather than just learn a rough fingerprint, they will still need to examine each individual URL regardless of how the side channel can operate.

Fermata cannot provably *prevent* a timing side channel from operating; it can only constrain the rate of bits transmitted across the channel. For any side channel it is important to consider the attacker’s goals along with how the side channel operates to understand what level of mitigation Fermata will provide. There are multiple reasons (compression, prior knowledge, etc.) that might lead to a side channel exhibiting behavior like described above. In all of these cases Fermata provides the same guarantee about channel bandwidth.

4.5 Isolating JavaScript from the world

A potential solution for JavaScript is to remove all run-to-completion violations, effectively ensuring that JavaScript cannot observe any state changes to the DOM or otherwise during a single execution. This necessarily includes all realtime clock accesses, as well as any other discovered exitless clocks. Since JavaScript will always have access to a fine grained minor clock (the `for` loop), it is critical that all exitless *major* clocks be removed. In the case of `performance.now`, this will result in the feature becoming an exiting clock, requiring that JavaScript stop execution before the available clock value changes.

The catch of the latter method is in how to remove all potential exitless clocks. If the upcoming SharedArrayBuffer API becomes available, this presents a highly accurate exitless clock that Fermata cannot mitigate without returning it to a message passing interface. Removing all of these potential exitless clocks requires an examination of all interfaces the JavaScript runtime has.

With all exitless clocks removed, the design need only focus on degrading exiting clocks to meet the target maximum channel bandwidth.

4.6 Degrading explicit clocks

Explicit clocks (ex: `performance.now`, `Date`, etc.) are degraded to some granularity g and update unpredictably. As in Hu [10], we accomplish this by performing updates to the clock value (at the granularity g) at randomized intervals. g is a multiple of the native OS time grain g_n (generally 1ns). Each randomized interval is a “tick,” during which the available explicit clocks do not change. At the beginning of each tick, we up-

date the Fermata clock to the rounded-down wallclock. Since the tick duration is not the same as g , the Fermata clocks will not always change in value every tick. This design guarantees that the available explicit clocks are only ever behind and are behind by a bounded amount of time, $g - g_n + (g/2)$. Note that a clock’s granularity does not alone define the accuracy to which it can be used to time some event, as seen with section 2.

Tick duration is not constant but is instead drawn from a uniform distribution with a mean of $g/2$. If intervals were constant and thus clock updates occurred exactly on the grain, the attacker could use the same clock-edge technique as in section 2.

4.7 Delaying events

The randomized update intervals (ticks) are further divided into alternating upticks and downticks for the purposes of delaying events and I/O. This mimics their usage in Hu [10]. Downticks cause outbound queued events to be flushed, and upticks cause inbound events to be delivered.

4.8 Tuning Fermata

Since the defensive guarantee provided by Fermata is only a maximum channel bandwidth, a few users may want to change the tradeoff between responsiveness and privacy. Fermata will provide this option via a tunable privacy setting that allows setting the acceptable leaking channel bandwidth. In turn, this will modify the average tick duration and the explicit time granularity, both of which affect usability. We expect that only developers (including of browser forks like Tor Browser) or users with specific privacy needs would interact with these settings.

5 Fuzzyfox prototype implementation

In this section we describe *Fuzzyfox*⁴, a prototype implementing many of the principles of the Fermata design in Mozilla Firefox. Fuzzyfox is not a complete Fermata solution but does show that the removal of exitless clocks and the delaying of events is a feasible design strategy for a browser.

Fuzzyfox attempts to mitigate the clocks of sections 2 and 3 by using the ideas in Fermata. Web browsers have an interest in degrading clocks available to JavaScript to reduce the impact of both known and unknown timing channel attacks. Fuzzyfox is a concrete demonstration of techniques that will make a browser more resistant to such timing attacks. As in Fermata, Fuzzyfox has a clock grain setting (g) and an average tick duration ($t_a = g/2$). All explicit clocks in Fuzzyfox report multiples of g .

We will refer to Firefox when discussing default behavior and Fuzzyfox when discussing the changes made.

5.1 Why Fuzzyfox?

We built Fuzzyfox for three reasons:

1. Building a new web browser is a monumental task.
2. We did not know if a Fermata-style design would result in a usable experience. It was entirely possible that the delays induced would render any Fermata-style designs unusable.
3. We want to deploy the insights of channel bandwidth mitigation to real systems like Tor Browser.

Fuzzyfox does *not* have the complete auditability advantages that Fermata would. However, we believe that our insights about principled fuzzing of explicit clocks can be directly applied to Tor Browser as an improvement to their ongoing efforts.

5.2 PauseTask

The core of the Fuzzyfox implementation is the PauseTask, a recurring event on the main thread event queue. The PauseTask provides two primary functions: it implicitly divides the execution of the event queue into discrete intervals, and it serves as the arbiter of uptick and downtick events.

Once Firefox has begun queuing events on the event queue, Fuzzyfox ensures that the first PauseTask gets added to the queue. From this point on, there will always be exactly one PauseTask on the event queue.

PauseTask does the following on each execution: determines remaining duration, generates retroactive ticks, sleeps remaining duration, updates clocks, flushes queues, and queues the next PauseTask.

Determine remaining duration

The PauseTask checks the current OS realtime clock (T_1) with microsecond accuracy using `gettimeofday`. Comparing this against the expected time between ticks (D_e) and the end of the last PauseTask (T_2) gives the actual duration (D_a). If $D_a \leq D_e$, PauseTask skips directly to sleeping away the remaining duration, $D_e - D_a$.

Optional: Retroactive ticks

Otherwise, PauseTask must retroactively generate the upticks and downticks that should have occurred. This ensures that even by being long running JavaScript cannot force a 0 sleep duration PauseTask.

Sleep remaining duration

PauseTask finishes out the remaining duration via `usleep`. `usleep` is not perfectly accurate, and has a fixed overhead cost. In our testing, `usleep` error varies based on the duration but is never enough to be an issue for Fuzzyfox.

Update all system clocks and flush queues

PauseTask now generates the new canonical system time. This is accomplished by taking the OS realtime clock and rounding down to the Fuzzyfox clock grain setting.

There are two underlying explicit time sources available to JavaScript, `Time` and `performance`. `PauseTask` directly updates the canonical `TimeStamp` time, which is used by `performance`, and delivers a message to the JavaScript runtimes to update `Time`'s canonical time. Our review found that all of the other time sources we knew of used `TimeStamp`.

In our prototype, the only I/O queue that needs to be flushed is the `DelayChannelQueue` (see section 5.3.) This only occurs if the currently executing `PauseTask` is a downtick.

Queue next PauseTask event

Finally, `PauseTask` queues the next `PauseTask` on the event queue. This sets the start time (T_1), marks the new `PauseTask` as either uptick or downtick, as well as drawing a random duration from the uniformly random distribution between 1 to $2 \times t_a$. `PauseTasks` are queued exclusively on the main thread to ensure they block JavaScript execution as well as all DOM manipulation events.

5.3 Queuing

All events visible to JavaScript must be queued in Fuzzyfox. Unfortunately, there is not a singular place or even explicit queues available for all events in Firefox. We use `PauseTask` to create implicit queues for all main thread events (including JavaScript callbacks, all DOM manipulations, all animations, and others) and construct our own queuing for network connections.

Timer events (including CSS animations, `setTimeout`, etc.) do not need to be explicitly modified from Firefox behavior, as they run in a separate thread that checks when timers should fire based on `TimeStamp`. As Fuzzyfox ensures all `TimeStamp`s are set to our canonical Fuzzyfox time, this is not a problem.

DelayChannelQueue

We implemented a simple arbitrary length queue for outgoing network connections called `DelayChannelQueue`. This queue contains any channels that have started to open and stops them from connecting to their external resource. In the Fuzzyfox prototype, we only queue outgoing HTTP requests, although it could easily be extended to more channel types. Upon receiving a downtick notification from `PauseTask`, the queue is locked and all currently queued channel connections are completed and flushed from the queue.

6 Fuzzyfox evaluation

We evaluated our prototype Fuzzyfox in both effectiveness (how it degrades clocks) and performance.

All evaluations are compared against a clean Firefox build without the Fuzzyfox patches. Firefox trunk⁵ was used as the basis and built with default build settings. Fuzzyfox patches are then applied on top of this commit and built with the same configuration. All tests were performed on an updated Ubuntu 14.04 machine with an Intel i5-4460 and 14GB of RAM. The only applications running during testing were the XFCE window manager and Fuzzyfox. Fuzzyfox and Firefox were both tested using the experimental e10s Firefox architecture. NSPR logging was enabled to capture data about Fuzzyfox internals.

6.1 Limitations

Fuzzyfox is not a complete Fermata implementation and is unable to guarantee a maximum channel bandwidth. Since we did not isolate the JavaScript engine from the DOM or all I/O operations, we did not interpose on all interfaces as would be required in a Fermata implementation. This is purely a practical decision, as accomplishing this in Firefox would require manually auditing the entire codebase. We do not, for example, interpose on synchronous IPC calls from JavaScript. See section 6.2.3 for an example of how this can break the Fermata guarantees.

Unfortunately, since our `PauseTasks` can be delayed by long running JavaScript on the main thread, we can no longer bound the difference between the OS realtime clock and the available explicit clocks. We do still guarantee that all explicit clocks are only ever behind realtime.

While we experimented with a number of different grain settings, the settings providing very high privacy guarantees (100s of milliseconds) have severe usability impact. We believe that a clean Fermata implementation may not incur such a strong usability impact at similar grain settings.

6.2 Effectiveness

Effectiveness is measured as the available resolution for a given clock. In the ideal case, all clocks in Fuzzyfox should be degraded provide a resolution no less than g . We measure the observed properties of the clocks described in section 3 between Firefox and Fuzzyfox. We set the explicit time granularity (g) to 100ms and the average `PauseTask` interval (t_a) to 50ms for these tests. We chose $g = 100ms$ because a large g value most clearly illustrates the difference between Fuzzyfox and Firefox. See section 6.3 for an evaluation of the impact of high g values on performance.

The following figures show scatter plots for several

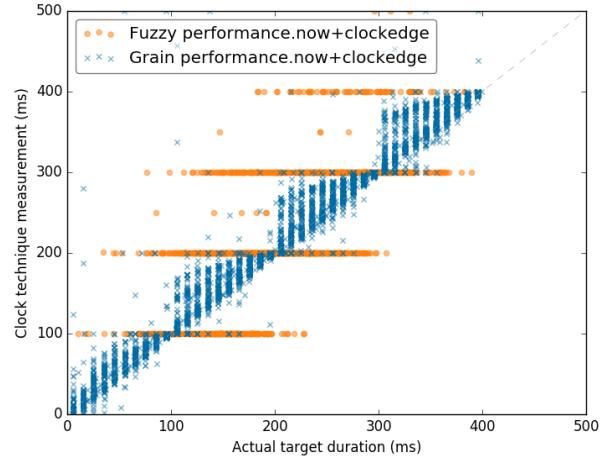


Figure 12: `performance.now` measurements with clock-edge on Fuzzyfox (existing) and Firefox (exitless, 100ms grain)

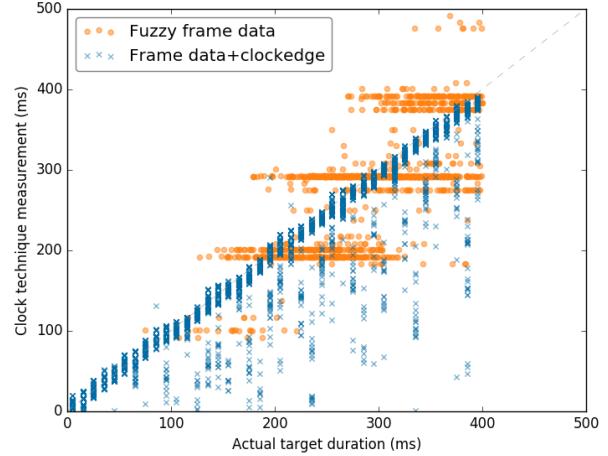


Figure 13: Frame data clock measurements on Firefox and Fuzzyfox

clock techniques as they operate in Firefox and in Fuzzyfox. In each, a perfectly accurate clock would follow the dashed grey line on $x = y$. Note that these figures show actual duration and clock technique duration, rather than target duration and error as in section 3.3. This is due to Fuzzyfox being unable to dependably schedule targets less than g (100ms) in duration. Thus, while the same testing code was used in Fuzzyfox and in Firefox, the actual durations of events are much longer in Fuzzyfox. Finally, there are no exitless clocks that we know of in Fuzzyfox to test, which would have been a closer comparison.

6.2.1 `performance.now`

Since time no longer moves forward during JavaScript execution, `performance.now` is now an existing

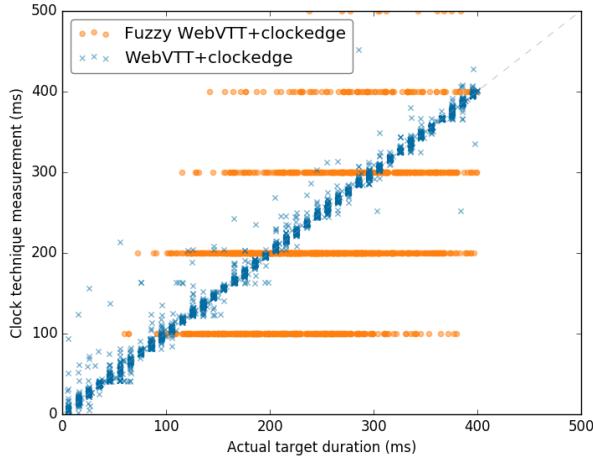


Figure 14: WebVTT clock measurements on Firefox and Fuzzyfox

clock. Figure 12 shows the results of using the clock-edge technique on `performance.now` for both Fuzzyfox and Firefox with a grain set to 100ms. Notably, clock-edge no longer improves the accuracy of the measurements! This demonstrates that the Fuzzyfox model successfully degrades explicit clocks.

6.2.2 Video frame data

Unexpectedly, Fuzzyfox transforms the video frame data clock from exitless to exiting. This is probably because the frame extracted for canvas is determined using the current explicit clock values (`TimeStamp`). Since time does not move forward during JavaScript execution, frame data is now an exiting clock. In general, we expect that run-to-completion violations (and by extension most exitless clocks) would not be properly degraded by Fuzzyfox. Figure 13 shows the exiting frame data clock on Fuzzyfox and Firefox.

6.2.3 WebSpeech API

Fuzzyfox degrades the WebSpeech API only because the `elapsedTime` field is drawn using the explicit clocks in Fuzzyfox. The starting and stopping of the speech is still synchronous, so it is possible some other piece of information passed back by the speech synthesis provider could provide a more accurate clock. WebSpeech should not be considered properly isolated by Fuzzyfox. Only if the *starting* and *stopping* of speech synthesis were queued like other events would Fuzzyfox correctly handle WebSpeech.

6.2.4 `setTimeout`

As `setTimeout` events are fired from the timer thread based on the degraded explicit clocks, they are no longer able to fire more often than the explicit time grain g of 100ms.

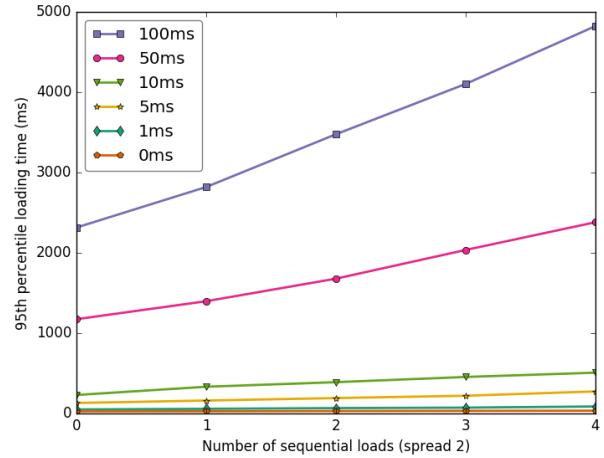


Figure 15: Page load times with variable depth for all Fuzzyfox configurations at a spread of 2

```
var njs=document.createElement('script')
njs.setAttribute('type','text/javascript')
njs.setAttribute('src','layer2.js')
document.getElementsByTagName('head')[0].appendChild(njs)
```

Figure 16: Iterative page load JavaScript

6.2.5 CSS Animations

As with `setTimeout`, CSS animation events are fired from the timer thread based on the degraded explicit clocks. Thus, they too are not able to be used as a clock of finer grain than the explicit time grain g .

6.2.6 XMLHttpRequests

XMLHttpRequests are properly degraded by Fuzzyfox. Since the callbacks for `onprogress` are queued on the main event queue and then gated by `PauseTask`, they are no longer timely when processed.

6.2.7 WebVTT subtitles

We examined the WebVTT subtitle implicit exiting clock in detail, as it performed among the best with the clock-edge technique on vanilla Firefox. Figure 14 shows the results for the same WebVTT clock techniques as described in section 3.2.2 on both Fuzzyfox and Firefox. Note that the clockedge code provided no benefits to the Fuzzyfox case.

6.3 Performance

Performance impact is difficult to measure, as most performance tools for browsers rely on accurate time measurements via JavaScript.

We performed a series of page load time tests, which show predictable results. We measure the impact of both *depth* of page loads and the *spread* of initial requests.

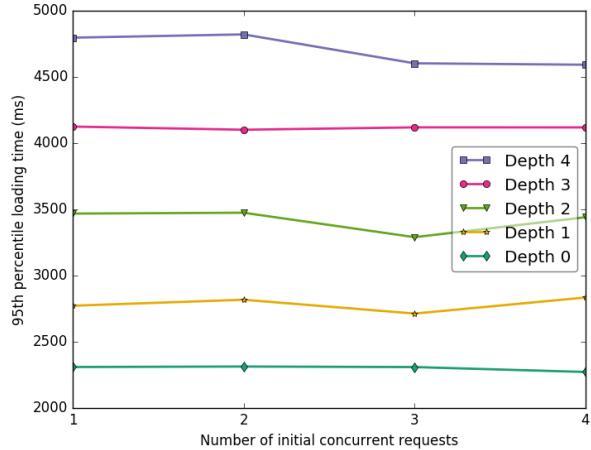


Figure 17: Page load times with variable spread and depth for $g = 100ms$

Our testing setup consisted of 20 test pages and 5 different fuzzyfox/Firefox configurations. The depth of the test pages represents how many sequential requests are made. Each request consists of inserting a script file of the form in figure 16. Each one has the loaded script be the next “layer” down, with layer 0 being an empty script. Thus, a test page that is 3 deep makes 4 sequential requests: `page.html`, `layer2.js`, `layer1.js`, `layer0.js`. Spread is achieved by the base `page.html` performing several duplicate initial requests to the top layer. Thus, a spread of 2 and a depth of 2 results in requests for: `page.html`, `layer1.js`, `layer1.js`, `layer0.js`, `layer0.js`. After the final page load completes, the total time from initial page navigation until completion is stored, and this process is repeated 1000 times per page test. We generate 20 test pages by combining up to 5 layers of depth with a spread from 1 to 5. We served the test pages via a basic nginx configuration running on the same host as the browser.

Figures 15 and 17 show two different views of some of the results, with the 95th percentile of load times being shown for $g = 100ms$. As expected, increasing the spread for a given depth (as shown in figure 17) results in almost no change to load times. All other browser configurations (see figure 18 for $g = 5ms$) had nearly identical results, with differing y-intercepts based on g . This occurs because outgoing HTTP requests in Fuzzyfox are batched, so queuing multiple requests at once does not incur any g -scaled penalties. However, as figure 15 shows, increasing the depth incurs a linear overhead with the slope and intercept scaled by the value of g . The worst case for Fuzzyfox are pages that do large numbers of sequential loads, each requiring JavaScript to run before the next load can be queued. Unfortunately, many modern webpages end up performing repeated loads of

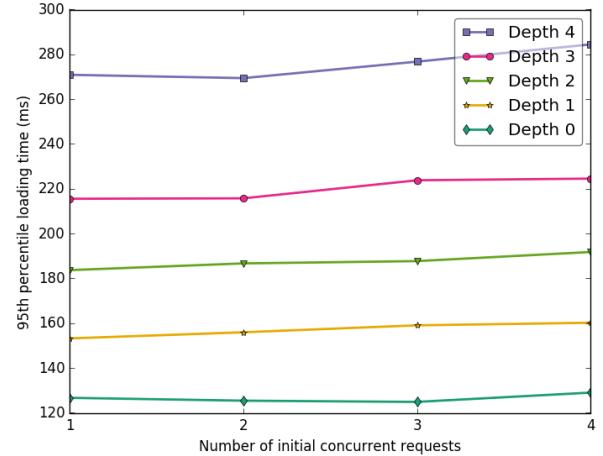


Figure 18: Page load times with variable spread and depth for $g = 5ms$

various libraries and partial content. One potential solution would be more widespread use of HTTP2’s Server Push which would alleviate the repeated g scaled penalties for resource requests.

JavaScript engine tests, such as JetStream, reported identical scores of 181 for both Firefox and Fuzzyfox.⁶ Fuzzyfox predictably records a maximum FPS equal to the average `PauseTask` fire rate or 20 FPS for $g = 100ms$, as compared to 60 FPS in the Firefox case.

6.3.1 Tor Browser

We also ran our page load tests on vanilla Tor Browser⁷. Rather than access the pages over the localhost interface, they are accessed over the Tor network. No other changes to the test setup were made. Due to the major changes in routing, the load times we observed are far more variable than in the Firefox or Fuzzyfox case and show no significant trends on the whole. If we compare the range of page load times between Fuzzyfox ($g = 100ms$) and Tor Browser in figures 19 and 20, we see that Tor Browser imposes a significantly *higher* overhead most of the time in both initial page load and in page load completion. Other spread levels show similar behavior. As in previous figures we show the 95th percentile load completion times but we additionally show the range from the minimum completion (`onload` fires) time as a shaded region.

6.3.2 Real world page loads

Table 3 shows a rough macro-benchmark of real-world page load times for Firefox, Fuzzyfox (various grains), and Tor Browser. In each case, the same Google search results page was loaded. These tests were manually performed and the reported page load time comes from the Firefox developer tools. Each load requested between

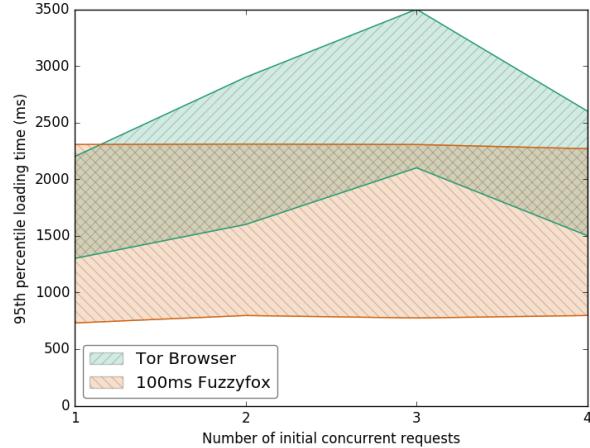


Figure 19: Range of page load completion times with variable depth at a spread of 0 for Tor Browser and Fuzzyfox $g = 100ms$

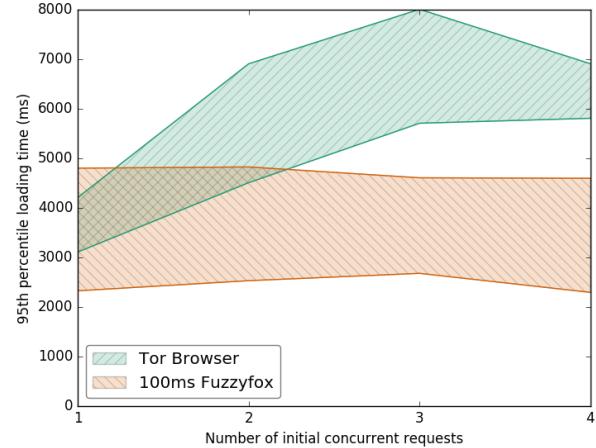


Figure 20: Range of page load completion times with variable depth at a spread of 4 for Tor Browser and Fuzzyfox $g = 100ms$

Browser or Grain(ms)	Reported load time(s)	
	Reload	Force Reload
Firefox	0.82	0.86
0.5	0.84	0.79
1	0.85	0.85
5	0.94	0.94
10	1.03	1.04
50	2.09	1.71
100	2.86	2.60
Tor	3.78	7.18

Table 3: Average page load times for https://www.google.com/?gws_rd=ssl#q=test+search with 10 reloads and 10 force reloads (no caching) on Firefox, Fuzzyfox, and Tor Browser

9 and 12 resources. The “force reload” column corresponds to a cache-less reload of the page, whereas the “reload” column indicates the load time with caching allowed. Minor differences between the reload and force reload results for a given browser are not statistically significant as we only have 10 samples.

While a larger study of more real-world pages would be valuable, such a study is larger in scope than this paper can cover. To perform such a measurement, we would need to individually determine a “load complete” point for each test page and re-instrument Fuzzyfox to enable measurements at these exact points. Google search results were chosen specifically because they do not continue to load resources indefinitely as many major websites do. (Ex: nytimes.com, youtube.com, etc.) We therefore leave a more detailed real-world page load time and user experience impact study to future work.

These metrics are incomplete, as they do not measure

interactivity of the pages, which can suffer in the Fuzzyfox case more than in Tor Browser. We leave further analysis of various performance impacts to future work.

While higher g settings cause significant page load time increases, these overheads are acceptable to some privacy conscious users and developers as demonstrated by Tor Browser. We do not have metrics for the impact of using both Tor Browser and our Fuzzyfox patch set, but we expect the overheads to be additive in the worst case. One option for integration with Tor Browser specifically would be to tune the value of g based on the setting of the “security slider” [20].

In light of these metrics, a g setting of $g \leq 5ms$ is likely tolerable for average use cases, while higher settings (up to and including $g = 100ms$) would likely be tolerated by users of Tor Browser. Ideally the clock fuzzing and other features as appropriate will be deployed in Firefox, and can be configured for a higher g in Tor Browser. If a more complete version of Fermata is developed, it will be worthwhile to run user studies before deploying g settings.

7 Related work

Popek and Kline [21] were the first to observe that the presence of clocks opens covert channels. They suggested that virtual machines be presented only with virtual clocks, not “a real time measure.” Lipner [16] responded that keeping virtual machines from correlating virtual time to real time is a “difficult problem,” since time is “the one system-wide resource [...] that can be observed in at least a coarse way by every user and every program.” Lipner suggested “randomizing the relation of virtual and real time” to add noise to the channel. Lipner also reported private communication from Saltzer

that timing channels had been demonstrated in Multics by mid-1975.

Digital’s VAX VMM Security Kernel project(initiated in 1981 and canceled in 1990 before its evaluation at the A1 level could be completed [12]) was the first system to attempt to randomize the relationship of virtual and real time. The VAX VMM Security Kernel team published three important papers describing their system. The first, by Karger et al. [11, 12], gave an overview of the system. The second, by Wray [28], presented a theory of time (“[w]e view the passage of time as being characterized by a sequence of events which can be distinguished one from another by an observer”) and of timing channels and is the source for our view, in this paper, of timing channels as arising from the comparison of a reference clock with a modulated clock. Wray noted that a process that increments a variable in a loop can be used as a clock. The third, by Hu [9, 10], described the VAX VMM’s fuzzy time system and is the inspiration for our paper. (A 2012 retrospective [15], though not the contemporaneous papers, reveals that the fuzzy time idea was developed in collaboration with the National Security Agency’s Robert Morris.) We describe many of the details of the fuzzy time system elsewhere in the paper. The 1992 journal version [9] of Hu’s paper gives a more complete security analysis than does the 1991 conference version [10]. In particular, it notes that fuzzy time would be defeated if the VM could devote a processor thread to incrementing a counter in memory shared with its other processor threads. This attack did not affect the Vax VMM Security Kernel, since it limited virtual machines to a single processor and did not support shared memory; it would apply to browsers if the proposed Shared Memory and Atomics specification [8] is implemented.

Several followup papers examined the security of fuzzy time. Trostle [25] observed that if scheduler time quanta coincide with upticks and if the scheduler employs a simple FIFO policy, then the scheduler can be used as a covert channel with 50 bps channel capacity. To send a bit, a high process either takes its entire time quantum or yields the processor; low processes try to send messages to each other in each time quantum. Which and how many messages arrived reveals the high process’ bit. Gray showed attacks on fuzzy time that exploit bus contention [7] and calculated a channel capacity for shared buses under fuzzy time under the assumption (satisfied in the case of the VAX VMM Security Kernel) that a low receiver can immediately notify the high sender when it receives an uptick [5]. A later tech report combines both papers by Gray [6].

Martin et al. [17] translated fuzzy time to the microarchitectural setting, proposing and evaluating a new microarchitecture in which execution is divided into variable-length “epochs.” The `rdtsc` instruction delays

execution until the next epoch and returns a cycle count randomly chosen from the last epoch. Because their focus is microarchitectural timing channels, Martin et al. argue that other sources of time, such as interrupt delivery, are inherently too coarse grained to need fuzzing. Martin et al. observe that simply rounding `rdtsc` to some granularity would be susceptible to clock-edge effects.

The success of infrastructure-as-a-service cloud computing brought with it the risk of cross-VM side channels [22]. Aviram et al. [2] proposed to close timing channels in cloud computing by enforcing deterministic execution and experimented with compiling a Linux kernel and userland not to use high-resolution timers like `rdtsc`, observing a drop in throughput. Vattikonda et al. [26] showed that it is possible to virtualize `rdtsc` for Xen guests, reducing its resolution (but allowing clock-edge attacks). Ford [4] proposed timing information flow control, or TIFC, “an extension of DIFC for reasoning about [...] the propagation of sensitive information into, out of, or within a software system via timing channels,” and proposed two mechanisms for implementing TIFC: deterministic execution and “pacing queues,” which are an extension of the VAX VMM Security Kernel’s interrupt queue mechanism.

Li et al. [13, 14] describe StopWatch, a virtual machine manager designed to defeat timing side channel attacks. In StopWatch, clocks are virtualized to “a deterministic function of the VM’s instructions executed so far”; multiple replicas of each VM are run in lockstep, and I/O timing for all of them is determined by the (virtual) time observed by the median replica.

Finally, Wu et Al. [29] present Deterland, a hypervisor that runs legacy operating systems deterministically. Deterland splits time into ticks and allows I/O only on tick boundaries. As in StopWatch, virtual time in Deterland is a function of the number of instructions executed.

8 Conclusions and future work

Restricting or removing timing side channels is a complex task. Simple degradation of available explicit clocks is an insufficient solution, allowing clock-edge techniques and implicit clocks to obtain additional timing information.

By drawing upon the lessons learned from trusted operating systems literature, we believe that browsers can be architected to mitigate all possible timing side channels. We propose Fermata as a design goal for such a verifiably resistant browser. Our Fuzzyfox patches to Firefox show that a Fermata-like design can intelligently make tradeoffs between performance and security, while not breaking the current interactions with JavaScript. Fuzzyfox empirically degrades clocks in a way that is

not susceptible to clock-edge techniques, protecting timing information.

Fuzzyfox requires a number of engineering improvements before it is ready to deploy to users, but it has proved that the fuzzy time concept can be applied to browsers. Notably, more experiments with setting channel bandwidth and exposing such settings to users need to be performed. Additionally, Fuzzyfox does not hook inbound network events, which a cooperating server could use to derive the duration of events in Fuzzyfox. Other interfaces (WebSockets, WebAudio, other media APIs) should be investigated for behavior that would break the Fuzzyfox design. We expect that with these changes Fuzzyfox could be adapted for use in projects like Tor Browser and protect real users against timing attacks.

Acknowledgements

We thank Kyle Huey, Patrick McManus, Eric Rescorla, and Martin Thomson at Mozilla for helpful discussions about this work, and for sharing their insights with us about Firefox internals. We are also grateful to Keaton Mowery and Mike Perry for helpful discussions, and to our anonymous reviewers and to David Wagner, our shepherd, for their detailed comments.

We additionally thank Nina Chen for assistance with editing and graph design.

This material is based upon work supported by the National Science Foundation under Grants No. 1228967 and 1514435, and by a gift from Mozilla.

References

- [1] M. Andryscy, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, “On subnormal floating point and abnormal timing,” in *Proceedings of IEEE Security and Privacy (“Oakland”) 2015*, L. Bauer and V. Shmatikov, Eds. IEEE Computer Society, May 2015.
- [2] A. Aviram, S. Hu, B. Ford, and R. Gummadi, “Determining timing channels in compute clouds,” in *Proceedings of CCSW 2010*, A. Perrig and R. Sion, Eds. ACM Press, Oct. 2010.
- [3] D. Cock, Q. Ge, T. Murray, and G. Heiser, “The last mile: An empirical study of timing channels on seL4,” in *Proceedings of CCS 2014*, M. Yung and N. Li, Eds. ACM Press, Nov. 2014, pp. 570–81.
- [4] B. Ford, “Plugging side-channel leaks with timing information flow control,” in *Proceedings of HotCloud 2012*, R. Fonseca and D. Maltz, Eds. USENIX, Jun. 2012.
- [5] J. W. Gray, “On analyzing the bus-contention channel under fuzzy time,” in *Proceedings of CSFW 1993*, C. Meadows, Ed. IEEE Computer Society, Jun. 1993, pp. 3–9.
- [6] ——, “Countermeasures and tradeoffs for a class of covert timing channels,” Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS94-18, 1994, online: <http://hdl.handle.net/1783.1/25>.
- [7] ——, “On introducing noise into the bus-contention channel,” in *Proceedings of IEEE Security and Privacy (“Oakland”) 1993*, R. Kemmerer and J. Rushby, Eds. IEEE Computer Society, May 1993, pp. 90–98.
- [8] L. T. Hansen, “ECMAScript shared memory and atomics,” Online: http://tc39.github.io/ecmascript_sharedmem/shmem.html, Feb. 2016.
- [9] W.-M. Hu, “Reducing timing channels with fuzzy time,” *J. Computer Security*, vol. 1, no. 3-4, pp. 233–54, 1992.
- [10] ——, “Reducing timing channels with fuzzy time,” in *Proceedings of IEEE Security and Privacy (“Oakland”) 1991*, T. F. Lunt and J. McLean, Eds. IEEE Computer Society, May 1991, pp. 8–20.
- [11] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, “A VMM security kernel for the VAX architecture,” in *Proceedings of IEEE Security and Privacy (“Oakland”) 1990*, D. M. Cooper and T. F. Lunt, Eds. IEEE Computer Society, May 1990, pp. 2–19.
- [12] ——, “A retrospective on the VAX VMM security kernel,” *IEEE Trans. Software Engineering*, vol. 17, no. 11, pp. 1147–65, Nov. 1991.
- [13] P. Li, D. Gao, and M. K. Reiter, “Mitigating access-driven timing channels in clouds using StopWatch,” in *Proceedings of DSN 2013*, G. Candea, Ed. IEEE/IFIP, Jun. 2013.
- [14] ——, “StopWatch: A cloud architecture for timing channel mitigation,” *ACM Trans. Info. & System Security*, vol. 17, no. 2, Nov. 2014.
- [15] S. Lipner, T. Jaeger, and M. E. Zurko, “Lessons from VAX/SVS for high-assurance VM systems,” *IEEE Security & Privacy*, vol. 10, no. 6, pp. 26–35, Nov.–Dec. 2012.
- [16] S. B. Lipner, “A comment on the confinement problem,” *ACM SIGOPS Operating Systems Review*, vol. 9, no. 5, pp. 192–96, Nov. 1975.
- [17] R. Martin, J. Demme, and S. Sethumadhavan, “Time-Warp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks,” in *Proceedings of ISCA 2012*, J. Torrellas, Ed. ACM Press, Jun. 2012, pp. 118–29.
- [18] Mozilla, “Javascript concurrency model and event loop,” 2016, online: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Run-to-completion>.
- [19] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, “The spy in the sandbox: Practical cache attacks in JavaScript and their implications,” in *Proceedings of CCS 2015*, C. Kruegel and N. Li, Eds. ACM Press, Oct. 2015.
- [20] M. Perry, “Tor browser 4.5 is released,” Apr. 2015, online: <https://blog.torproject.org/blog/tor-browser-45-released>.

- [21] G. J. Popek and C. S. Kline, “Verifiable secure operating system software,” in *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*. ACM, May 1974, pp. 145–51.
- [22] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds,” in *Proceedings of CCS 2009*, S. Jha and A. Keromytis, Eds. ACM Press, Nov. 2009, pp. 199–212.
- [23] M. Seaborn, “Security: Chrome provides high-res timers which allow cache side channel attacks,” 2015, online: <https://bugs.chromium.org/p/chromium/issues/detail?id=508166>.
- [24] P. Stone, “Pixel perfect timing attacks with HTML5,” Presented at Black Hat 2013, Jul. 2013, online: http://contextis.co.uk/documents/2/Browser_Timing_Attacks.pdf.
- [25] J. T. Trostle, “Modelling a fuzzy time system,” in *Proceedings of IEEE Security and Privacy (“Oakland”)* 1993, R. Kemmerer and J. Rushby, Eds. IEEE Computer Society, May 1993, pp. 82–89.
- [26] B. C. Vattikonda, S. Das, and H. Shacham, “Eliminating fine grained timers in Xen (short paper),” in *Proceedings of CCSW 2011*, T. Ristenpart and C. Cachin, Eds. ACM Press, Oct. 2011.
- [27] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, “A practical attack to de-anonymize social network users,” in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 223–238.
- [28] J. C. Wray, “An analysis of covert timing channels,” in *Proceedings of IEEE Security and Privacy (“Oakland”)*
- 1991, T. F. Lunt and J. McLean, Eds. IEEE Computer Society, May 1991, pp. 2–7.
- [29] W. Wu, E. Zhai, D. I. Wolinsky, B. Ford, L. Gu, and D. Jackowitz, “Warding off timing attacks in Deterland,” in *Proceedings of TRIOS 2015*, L. Shrira, Ed. ACM Press, Oct. 2015.

Notes

¹https://bugzilla.mozilla.org/show_bug.cgi?id=711043

²<https://trac.torproject.org/projects/tor/ticket/1517>

³commit 0ec3174fe63d8139f842ce9eb6639349759ff4e5

⁴Fuzzyfox is available as a branch at <https://github.com/dkohlbrenner/gecko-dev>. It should be treated as an engineering prototype.

⁵Firefox tests were done with commit 0ec3174fe63d8139f842ce9eb6639349759ff4e5 for clock tests, and c4afaf3404986ccc1d221bc7f4f3f1dcf39b06fc for the page load tests

⁶Fuzzyfox was modified to report valid performance.now results for performance testing

⁷Tor Browser git revision: b60b8871fa08feaaca24bcf6dff43df0cd1c5f29 modified to report accurate performance.now values

Tracing Information Flows Between Ad Exchanges Using Retargeted Ads

Muhammad Ahmad Bashir
Northeastern University
ahmad@ccs.neu.edu

Sajjad Arshad
Northeastern University
arshad@ccs.neu.edu

William Robertson
Northeastern University
wkr@ccs.neu.edu

Christo Wilson
Northeastern University
cbw@ccs.neu.edu

Abstract

Numerous surveys have shown that Web users are concerned about the loss of privacy associated with online tracking. Alarmingly, these surveys also reveal that people are also unaware of the amount of data sharing that occurs between ad exchanges, and thus underestimate the privacy risks associated with online tracking.

In reality, the modern ad ecosystem is fueled by a flow of user data between trackers and ad exchanges. Although recent work has shown that ad exchanges routinely perform *cookie matching* with other exchanges, these studies are based on brittle heuristics that cannot detect all forms of information sharing, especially under adversarial conditions.

In this study, we develop a methodology that is able to detect client- and server-side flows of information between arbitrary ad exchanges. Our key insight is to leverage *retargeted ads* as a tool for identifying information flows. Intuitively, our methodology works because it relies on the *semantics* of how exchanges serve ads, rather than focusing on specific cookie matching *mechanisms*. Using crawled data on 35,448 ad impressions, we show that our methodology can successfully categorize four different kinds of information sharing behavior between ad exchanges, including cases where existing heuristic methods fail.

We conclude with a discussion of how our findings and methodologies can be leveraged to give users more control over what kind of ads they see and how their information is shared between ad exchanges.

1 Introduction

People have complicated feelings with respect to online behavioral advertising. While surveys have shown that some users prefer relevant, targeted ads to random, untargeted ads [60, 14], this preference has caveats. For example, users are uncomfortable with ads that are tar-

geted based on sensitive Personally Identifiable Information (PII) [44, 4] or specific kinds of browsing history (*e.g.*, visiting medical websites) [41]. Furthermore, some users are universally opposed to online tracking, regardless of circumstance [46, 60, 14].

One particular concern held by users is their “digital footprint” [33, 65, 58], *i.e.*, which first- and third-parties are able to track their browsing history? Large-scale web crawls have repeatedly shown that trackers are ubiquitous [24, 19], with DoubleClick alone being able to observe visitors on 40% of websites in the Alexa Top-100K [11]. These results paint a picture of a balkanized web, where trackers divide up the space and compete for the ability to collect data and serve targeted ads.

However, this picture of the privacy landscape is at odds with the current reality of the ad ecosystem. Specifically, ad exchanges routinely perform *cookie matching* with each other, to synchronize unique identifiers and share user data [2, 54, 21]. Cookie matching is a precondition for ad exchanges to participate in *Real Time Bidding* (RTB) auctions, which have become the dominant mechanism for buying and selling advertising inventory from publishers. Problematically, Hoofnagle *et al.* report that users naïvely believe that privacy policies prevent companies from sharing user data with third-parties, which is not always the case [32].

Despite user concerns about their digital footprint, we currently lack the tools to fully understand how information is being shared between ad exchanges. Prior empirical work on cookie matching has relied on heuristics that look for specific strings in HTTP messages to identify flows between ad networks [2, 54, 21]. However, these heuristics are brittle in the face of obfuscation: for example, DoubleClick cryptographically hashes their cookies before sending them to other ad networks [1]. More fundamentally, analysis of *client-side* HTTP messages are insufficient to detect *server-side* information flows between ad networks.

In this study, we develop a methodology that is able to detect client- and server-side flows of information between arbitrary ad exchanges that serve *retargeted ads*. Retargeted ads are the most specific form of behavioral ads, where a user is targeted with ads related to the exact products she has previously browsed (see § 2.2 for definition). For example, Bob visits nike.com and browses for running shoes but decides not to purchase them. Bob later visits cnn.com and sees an ad for the exact same running shoes from Nike.

Our key insight is to leverage retargeted ads as a mechanism for identifying information flows. This is possible because the strict conditions that must be met for a retarget to be served allow us to infer the precise flow of tracking information that facilitated the serving of the ad. Intuitively, our methodology works because it relies on the *semantics* of how exchanges serve ads, rather than focusing on specific cookie matching *mechanisms*.

To demonstrate the efficacy of our methodology, we conduct extensive experiments on real data. We train 90 *personas* by visiting popular e-commerce sites, and then crawl major publishers to gather retargeted ads [9, 12]. Our crawler is an instrumented version of Chromium that records the *inclusion chain* for every resource it encounters [5], including 35,448 chains associated with 5,102 unique retargeted ads. We use carefully designed pattern matching rules to categorize each of these chains, which reveal 1) the pair of ad exchanges that shared information in order to serve the retarget, and 2) the mechanism used to share the data (*e.g.*, cookie matching).

In summary, we make the following contributions:

- We present a novel methodology for identifying information flows between ad networks that is content- and ad exchange-agnostic. Our methodology allows to identify four different categories of information sharing between ad exchanges, of which cookie matching is one.
- Using crawled data, we show that the heuristic methods used by prior work to analyze cookie matching are unable to identify 31% of ad exchange pairs that share data.
- Although it is known that Google’s privacy policy allows it to share data between its services [26], we provide the first empirical evidence that Google uses this capability to serve retargeted ads.
- Using graph analysis, we show how our data can be used to automatically infer the roles played by different ad exchanges (*e.g.*, Supply-Side and Demand-Side Platforms). These results expand upon prior work [25] and facilitate a more nuanced understanding of the online ad ecosystem.

Ultimately, we view our methodology as a stepping stone towards more balanced privacy protection tools for

users, that also enable publishers to earn revenue. Surveys have shown that users are not necessarily opposed to online ads: some users are just opposed to tracking [46, 60, 14], while others simply desire more nuanced control over their digital footprint [4, 41]. However, existing tools (*e.g.*, browser extensions) cannot distinguish between targeted and untargeted ads, thus leaving users with no alternative but to block all ads. Conversely, our results open up the possibility of building in-browser tools that just block cookie matching, which will effectively prevent most targeted ads from RTB auctions, while still allowing untargeted ads to be served.

Open Source. As a service to the community, we have open sourced all the data from this project. This includes over 7K labeled behaviorally targeted and retargeted ads, as well as the inclusion chains and full HTTP traces associated with these ads. The data is available at:

<http://personalization.ccs.neu.edu/>

2 Background and Definitions

In this section, we set the stage for our study by providing background about the online display ad industry, as well as defining key terminology. We focus on techniques and terms related to Real Time Bidding and retargeted ads, since they are the focus of our study.

2.1 Online Display Advertising

Online display advertising is fundamentally a matching problem. On one side are *publishers* (*e.g.*, news websites, blogs, *etc.*) who produce content, and earn revenue by displaying ads to users. On the other side are advertisers who want to display ads to particular users (*e.g.*, based on demographics or market segments). Unfortunately, the online user population is fragmented across hundreds of thousands of publishers, making it difficult for advertisers to reach desired customers.

Ad networks bridge this gap by aggregating *inventory* from publishers (*i.e.*, space for displaying ads) and filling it with ads from advertisers. Ad networks make it possible for advertisers to reach a broad swath of users, while also guaranteeing a steady stream of revenue for publishers. Inventory is typically sold using a Cost per Mille (CPM) model, where advertisers purchase blocks of 1000 *impressions* (views of ads), or a Cost per Click (CPC) model, where the advertiser pays a small fee each time their ad is clicked by a user.

Ad Exchanges and Auctions. Over time, ad networks are being supplanted by *ad exchanges* that rely on an auction-based model. In Real-time Bidding (RTB) exchanges, advertisers bid on individual impressions, in real-time; the winner of the auction is permitted to serve

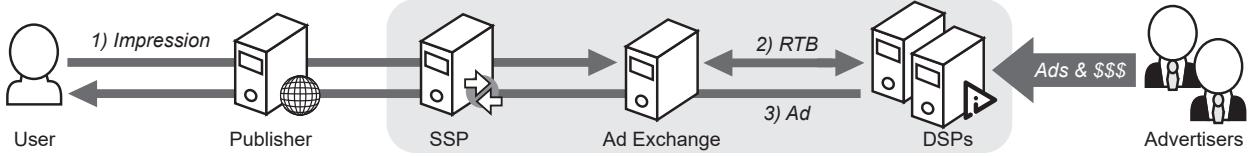


Figure 1: The display advertising ecosystem. Impressions and tracking data flow left-to-right, while revenue and ads flow right-to-left.

an ad to the user. Google’s DoubleClick is the largest ad exchange, and it supports RTB.

As shown in Figure 1, there is a distinction between Supply-side Platforms (*SSPs*) and Demand-side Platforms (*DSPs*) with respect to ad auctions. *SSPs* work with publishers to manage their relationships with multiple ad exchanges, typically to maximize revenue. For example, OpenX is an *SSP*. In contrast, *DSPs* work with advertisers to assess the value of each impression and optimize bid prices. MediaMath is an example of a *DSP*. To make matters more complicated, many companies offer products that cross categories; for example, Rubicon Project offers *SSP*, ad exchange, and *DSP* products. We direct interested readers to [45] for more discussion of the modern online advertising ecosystem.

2.2 Targeted Advertising

Initially, the online display ad industry focused on generic brand ads (*e.g.*, “Enjoy Coca-Cola!”) or *contextual ads* (*e.g.*, an ad for Microsoft on StackOverflow). However, the industry quickly evolved towards *behavioral targeted ads* that are served to specific users based on their browsing history, interests, and demographics.

Tracking. To serve targeted ads, ad exchanges and advertisers must collect data about online users by tracking their actions. Publishers embed JavaScript or invisible “tracking pixels” that are hosted by tracking companies into their web pages, thus any user who visits the publisher also receives third-party cookies from the tracker (we discuss other tracking mechanisms in § 3). Numerous studies have shown that trackers are pervasive across the Web [38, 36, 55, 11], which allows ad-

vertisers to collect users’ browsing history. All major ad exchanges, like DoubleClick and Rubicon, perform user tracking, but there are also companies like BlueKai that just specialize in tracking.

Cookie Matching. During an RTB ad auction, *DSPs* submit bids on an impression. The amount that a *DSP* bids on a given impression is intrinsically linked to the amount of information they have about that user. For example, a *DSP* is unlikely to bid highly for user u whom they have never observed before, whereas a *DSP* may bid heavily for user v who they have recently observed browsing high-value websites (*e.g.*, the baby site TheBump.com).

However, the Same Origin Policy (SOP) hinders the ability of *DSPs* to identify users in ad auctions. As shown in Figure 1, requests are first sent to an *SSP* which forwards the impression to an exchange (or holds the auction itself). At this point, the *SSP*’s cookies are known, but not the *DSPs*. This leads to a catch-22 situation: a *DSP* cannot read its cookies until it contacts the user, but it cannot contact the user without first bidding and winning the auction.

To circumvent SOP restrictions, ad exchanges and advertisers engage in *cookie matching* (sometimes called *cookie syncing*). *Cookie matching* is illustrated in Figure 2: the user’s browser first contacts ad exchange s .com, which returns an HTTP redirect to its partner d .com. s reads its own cookie, and includes it as a parameter in the redirect to d . d now has a mapping from its cookie to s ’s. In the future, if d participates in an auction held by s , it will be able to identify matched users using s ’s cookie. Note that some ad exchanges (including DoubleClick) send cryptographically hashed cookies to their partners, which prevents the ad network’s true cookies from leaking to third-parties.

Retargeted Ads. In this study, we focus on *retargeted ads*, which are the most specific type of targeted display ads. Two conditions must be met for a *DSP* to serve a retargeted ad to a user u : 1) the *DSP* must know that u browsed a specific product on a specific e-commerce site, and 2) the *DSP* must be able to uniquely identify u during an auction. If these conditions are met, the *DSP* can serve u a highly personalized ad reminding them to purchase the product from the retailer. Cookie



Figure 2: *SSP* s matches their cookie to *DSP* d using an HTTP redirect.

matching is crucial for ad retargeting, since it enables DSPs to meet requirement (2).

3 Related Work

Next, we briefly survey related work on online advertising. We begin by looking at more general studies of the advertising and tracking ecosystem, and conclude with a more focused examination of studies on cookie matching and retargeting. Although existing studies on cookie matching demonstrate that this practice is widespread and that the privacy implications are alarming, these works have significant methodological shortcomings that motivate us to develop new techniques in this work.

3.1 Measuring the Ad Ecosystem

Numerous studies have measured and broadly characterized the online advertising ecosystem. Guha *et al.* were the first to systematically measure online ads, and their carefully controlled methodology has been very influential on subsequent studies (including this one) [27]. Barford *et al.* take a much broader look at the *adscape* to determine who the major ad networks are, what fraction of ads are targeted, and what user characteristics drive targeting [9]. Carrascosa *et al.* take an even finer grained look at targeted ads by training *personas* that embody specific interest profiles (*e.g.*, cooking, sports), and find that advertisers routinely target users based on sensitive attributes (*e.g.*, religion) [12]. Rodriguez *et al.* measure the ad ecosystem on mobile devices [61], while Zarras *et al.* analyzed malicious ad campaigns and the ad networks that serve them [66].

Note that **none** of these studies examine retargeted ads; Carrascosa *et al.* specifically excluded retargets from their analysis [12].

Trackers and Tracking Mechanisms. To facilitate ad targeting, participants in the ad ecosystem must extensively track users. Krishnamurthy *et al.* have been cataloging the spread of trackers and assessing the ensuing privacy implications for years [38, 36, 37]. Roesner *et al.* develop a comprehensive taxonomy of different tracking mechanisms that store state in users' browsers (*e.g.*, cookies, HTML5 LocalStorage, and Flash LSOs), as well as strategies to block them [55]. Gill *et al.* use large web browsing traces to model the revenue earned by different trackers (or *aggregators* in their terminology), and found that revenues are skewed towards the largest trackers (primarily Google) [24]. More recently, Cahn *et al.* performed a broad survey of cookie characteristics across the Web, and found that less than 1% of trackers can aggregate information across 75% of websites in the Alexa Top-10K [11]. Falahrastegar *et al.* ex-

pand on these results by comparing trackers across geographic regions [20], while Li *et al.* show that most tracking cookies can be automatically detected using simple machine learning methods [42].

Note that **none** of these studies examine cookie matching, or information sharing between ad exchanges.

Although users can try to evade trackers by clearing their cookies or using private/incognito browsing modes, companies have fought back using techniques like *Evercookies* and *fingerprinting*. Evercookies store the tracker's state in many places within the browser (*e.g.*, FlashLSOs, etags, *etc.*), thus facilitating regeneration of tracking identifiers even if users delete their cookies [34, 57, 6, 47]. Fingerprinting involves generating a unique ID for a user based on the characteristics of their browser [18, 48, 50], browsing history [53], and computer (*e.g.*, the HTML5 canvas [49]). Several studies have found trackers in-the-wild that use fingerprinting techniques [3, 52, 35]; Nikiforakis *et al.* propose to stop fingerprinting by carefully and intentionally adding more entropy to users' browsers [51].

User Profiles. Several studies specifically focus on tracking data collected by Google, since their trackers are more pervasive than any others on the Web [24, 11]. Alarming, two studies have found that Google's Ad Preferences Manager, which is supposed to allow users to see and adjust how they are being targeted for ads, actually hides sensitive information from users [64, 16]. This finding is troubling given that several studies rely on data from the Ad Preferences Manager as their source of ground-truth [27, 13, 9]. To combat this lack of transparency, Lecuyer *et al.* have built systems that rely on controlled experiments and statistical analysis to infer the profiles that Google constructs about users [39, 40]. Castelluccia *et al.* go further by showing that adversaries can infer users' profiles by passively observing the targeted ads they are shown by Google [13].

3.2 Cookie Matching and Retargeting

Although ad exchanges have been transitioning to RTB auctions since the mid-2000s, only three empirical studies have examined the cookie matching that enables these services. Acar *et al.* found that hundreds of domains passed unique identifiers to each other while crawling websites in the Alexa Top-3K [2]. Olejnik *et al.* noticed that ad auctions were leaking the winning bid prices for impressions, thus enabling a fascinating behind-the-scenes look at RTB auctions [54]. In addition to examining the monetary aspects of auctions, Olejnik *et al.* found 125 ad exchanges using cookie matching. Finally, Falahrastegar *et al.* examine the clusters of domains that all share unique, matched cookies using crowdsourced browsing data [21]. Additionally, Ghosh *et al.* use game

theory to model the incentives for ad exchanges to match cookies with their competitors, but they provide no empirical measurements of cookie matching [23].

Several studies examine retargeted ads, which are directly facilitated by cookie matching and RTB. Liu *et al.* identify and measure retargeted ads served by DoubleClick by relying on unique AdSense tags that are embedded in ad URLs [43]. Olejnik *et al.* crawled specific e-commerce sites in order to elicit retargeted ads from those retailers, and observe that retargeted ads can cost advertisers over \$1 per impression (an enormous sum, considering contextual ads sell for <\$0.01) [54].

Limitations. The prior work on cookie matching demonstrates that this practice is widespread. However, these studies also have significant methodological limitations, which prevent them from observing all forms of information sharing between ad exchanges. Specifically:

1. All three studies identify cookie matching by locating unique user IDs that are transmitted to multiple third-party domains [2, 54, 21]. Unfortunately, this will miss cases where exchanges send permuted or obfuscated IDs to their partners. Indeed, DoubleClick is known to do this [1].
2. The two studies that have examined the behavior of DoubleClick have done so by relying on specific cookie keys and URL parameters to detect cookie matching and retargeting [54, 43]. Again, these methods are not robust to obfuscation or encryption that hide the content of HTTP messages.
3. Existing studies cannot determine the precise information flows between ad exchanges, *i.e.*, which parties are sending or receiving information [2]. This limitation stems from analysis techniques that rely entirely on analyzing HTTP headers. For example, a script from t1.com embedded in pub.com may share cookies with t2.com using dynamic AJAX, but the referrer appears to be pub.com, thus potentially hiding t1’s role as the source of the flow.

In general, these limitations stem from a reliance on analyzing specific *mechanisms* for cookie matching. In this study, one of our primary goals is to develop a methodology for detecting cookie matching that is agnostic to the underlying matching mechanism, and instead relies on the fundamental *semantics* of ad exchanges.

4 Methodology

In this study, our primary goal is to develop a methodology for detecting flows of user data between arbitrary ad exchanges. This includes client-side flows (*i.e.*, cookie matching), as well as server-side flows.

In this section, we discuss the methods and data we use to meet this goal. First, we briefly sketch our high-level approach, and discuss key enabling insights. Second, we introduce the instrumented version of Chromium that we use during our crawls. Third, we explain how we designed and trained shopper *personas* that view products on the web, and finally we detail how we collected ads using the trained personas.

4.1 Insights and Approach

Although prior work has examined information flow between ad exchanges, these studies are limited to specific types of cookie matching that follow well-defined patterns (see § 3.2). To study arbitrary information flows in a mechanism-agnostic way, we need a fundamentally different methodology.

We solve this problem by relying on a key insight: in most cases, if a user is served a retargeted ad, this proves that ad exchanges shared information about the user (see § 6.1.1). To understand this insight, consider that two preconditions must be met for user u to be served a retarget ad for *shop* by DSP d . *First*, either d directly observed u visiting *shop*, or d must be told this information by SSP s . If this condition is not met, then d would not pay the premium price necessary to serve u a retarget. *Second*, if the retarget was served from an ad auction, SSP s and d must be sharing information about u . If this condition is not met, then d would have no way of identifying u as the source of the impression (see § 2.2).

In this study, we leverage this observation to reliably infer information flows between SSPs and DSPs, regardless of whether the flow occurs client- or server-side. The high-level methodology is quite intuitive: have a clean browser visit specific e-commerce sites, then crawl publishers and gather ads. If we observe retargeted ads, we know that ad exchanges tracking the user on the *shopper-side* are sharing information with exchanges serving ads on the *publisher-side*. Specifically, our methodology uses the following steps:

- § 4.2: We use an instrumented version of Chromium to record *inclusion chains* for all resources encountered during our crawls [5]. These chains record the precise origins of all resource requests, even when the requests are generated dynamically by JavaScript or Flash. We use these chains in § 6 to categorize information flows between ad exchanges.
- § 4.3: To elicit retargeted ads from ad exchanges, we design *personas* (to borrow terminology from [9] and [12]) that visit specific e-commerce sites. These sites are carefully chosen to cover different types of products, and include a wide variety of common trackers.

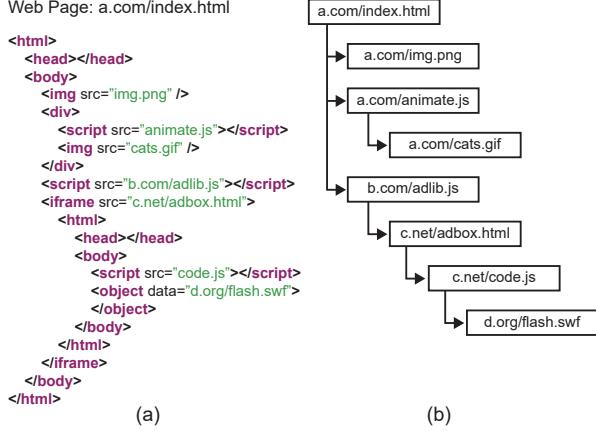


Figure 3: (a) DOM Tree, and (b) Inclusion Tree.

- § 4.4: To collect ads, our personas crawl 150 publishers from the Alexa Top-1K.
- § 5: We leverage well-known filtering techniques and crowdsourcing to identify retargeted ads from our corpus of 571,636 unique crawled images.

4.2 Instrumenting Chromium

Before we can begin crawling, we first need a browser that is capable of recording detailed information about the provenance of third-party resource inclusions in webpages. Recall that prior work on cookie matching was unable to determine which ad exchanges were syncing cookies in many cases because the analysis relied solely on the contents of HTTP requests [2, 21] (see § 3.2). The fundamental problem is that HTTP requests, and even the DOM tree itself, do not reveal the true sources of resource inclusions in the presence of dynamic code (JavaScript, Flash, etc.) from third-parties.

To understand this problem, consider the example DOM tree for a.com/index.html in Figure 3(a). Based on the DOM, we might conclude that the chain $a \rightarrow c \rightarrow d$ captures the sequence of inclusions leading from the root of the page to the Flash object from d.org.

However, direct use of a webpage’s DOM is misleading because the DOM does not reliably record the inclusion relationships between resources in a page. This is due to the ability of JavaScript to manipulate the DOM at run-time, *i.e.*, by adding new inclusions dynamically. As such, while the DOM is a faithful syntactic description of a webpage *at a given point in time*, it cannot be relied upon to extract relationships between included resources. Furthermore, analysis of HTTP request headers does not solve this problem, since the Referer is set to the first-party domain even when inclusions are dynamically added by third-party scripts.

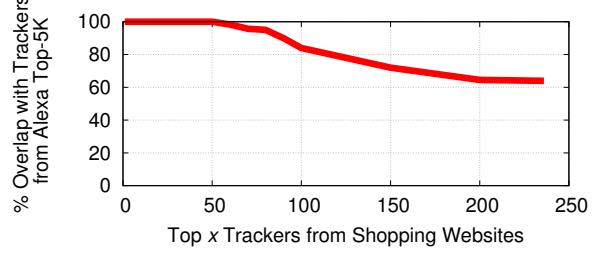


Figure 4: Overlap between frequent trackers on e-commerce sites and Alexa Top-5K sites.

To solve this issue, we make use of a heavily instrumented version of Chromium that produces *inclusion trees* directly from Chromium’s resource loading code [5]. Inclusion trees capture the semantic inclusion structure of resources in a webpage (*i.e.*, which objects cause other objects to be loaded), unlike DOM trees which only capture syntactic structures. Our instrumented Chromium accurately captures relationships between elements, regardless of where they are located (*e.g.*, within a single page or across frames) or how the relevant code executes (*e.g.*, via an inline `<script>`, `eval()`, or an event handler). We direct interested readers to [5] for more detailed information about inclusion trees, and the technical details of how the Chromium binary is instrumented.

Figure 3(b) shows the inclusion tree corresponding to the DOM tree in Figure 3(a). From the inclusion tree, we can see that the true *inclusion chain* leading to the Flash object is $a \rightarrow b \rightarrow c \rightarrow c \rightarrow d$, since the IFrame and the Flash are dynamically included by JavaScript from b.com and c.net, respectively.

Using inclusion chains, we can precisely analyze the provenance of third-party resources included in webpages. In § 6, we use this capability to distinguish client-side flows of information between ad exchanges (*i.e.*, cookie matching) from server-side flows.

4.3 Creating Shopper Personas

Now that we have a robust crawling tool, the next step in our methodology is designing shopper personas. Each persona visits products on specific e-commerce sites, in hope of seeing retargeted ads when we crawl publishers.

Since we do not know *a priori* which e-commerce sites are conducting retargeted ad campaigns, our personas must cover a wide variety of sites. To facilitate this, we leverage the hierarchical categorization of e-commerce sites maintained by Alexa¹. Although Alexa’s hierarchy

¹<http://www.alexa.com/topsites/category/Top/Shopping>

has 847 total categories, there is significant overlap between categories. We manually selected 90 categories to use for our personas that have minimal overlap, as well as cover major e-commerce sites (*e.g.*, Amazon and Walmart) and shopping categories (*e.g.*, sports and jewelry).

For each persona, we included the top 10 e-commerce sites in the corresponding Alexa category. In total, the personas cover 738 unique websites. Furthermore, we manually selected 10 product URLs on each of these websites. Thus, each persona visits 100 products URLs.

Sanity Checking. The final step in designing our personas is ensuring that the e-commerce sites are embedded with a representative set of trackers. If they are not, we will not be able to collect targeted ads.

Figure 4 plots the overlap between the trackers we observe on the Alexa Top-5K websites, compared to the top x trackers (*i.e.*, most frequent) we observe on the e-commerce sites. We see that 84% of the top 100 e-commerce trackers are also present in the trackers on Alexa Top-5K sites². These results demonstrate that our shopping personas will be seen by the vast majority of major trackers when they visit our 738 e-commerce sites.

4.4 Collecting Ads

In addition to selecting e-commerce sites for our personas, we must also select publishers to crawl for ads. We manually select 150 publishers by examining the Alexa Top-1K websites and filtering out those which do not display ads, are non-English, are pornographic, or require logging-in to view content (*e.g.*, Facebook). We randomly selected 15 URLs on each publisher to crawl.

At this point, we are ready to crawl ads. We initialized 91 copies of our instrumented Chromium binary: 90 corresponding to our shopper personas, and one which serves as a control. During each *round* of crawling, the personas visit their associated e-commerce sites, then visit the 2,250 publisher URLs (150 publishers * 15 pages per publisher). The control *only* visits the publisher URLs, *i.e.*, it does not browse e-commerce sites, and therefore should never be served retargeted ads. The crawlers are executed in tandem, so they visit the publishers URLs in the same order at the same times. We hard-coded a 1 minute delay between subsequent page loads to avoid overloading any servers, and to allow time for the crawler to automatically scroll to the bottom of each page. Each round takes 40 hours to complete.

We conducted nine rounds of crawling between December 4 to 19, 2015. We stopped after 9 rounds because we observed that we only gathered 4% new images during the ninth round. The crawlers recorded inclusion

²We separately crawled the resources included by the Alexa Top-5K websites in January 2015. For each website, we visited 6 pages and recorded all the requested resources.

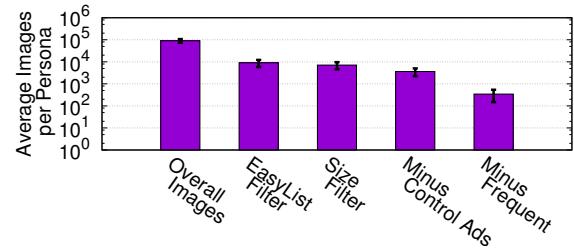


Figure 5: Average number of images per persona, with standard deviation error bars.

trees, HTTP request and response headers, cookies, and images from all pages. At no point did our crawlers click on ads, since this can be construed as click-fraud (*i.e.*, advertisers often have to pay each time their ads are clicked, and thus automated clicks drain their advertising budget). All crawls were done from *Northeastern University’s IP addresses in Boston*.

5 Image Labeling

Using the methodology in § 4.4, we collected 571,636 unique images in total. However, only a small subset are retargeted ads, which are our focus. In this section, we discuss the steps we used to filter down our image set and isolate retargeted ads, beginning with standard filters used by prior work [9, 42], and ending with crowd-sourced image labeling.

5.1 Basic Filtering

Prior work has used a number of techniques to identify ad images from crawled data. First, we leverage the *EasyList* filter³ provided by *AdBlockPlus* to detect images that are likely to be ads [9, 42]. In our case, we look at the inclusion chain for each image, and filter out those in which none of the URLs in the chain are a hit against EasyList. This reduces the set to 93,726 unique images.

Next, we filter out all images with dimensions $< 50 \times 50$ pixels. These images are too small to be ads; most are 1×1 tracking pixels.

Our final filter relies on a unique property of retargeted ads: they should only appear to personas that visit a specific e-commerce site. In other words, any ad that was shown to our control account (which visits no e-commerce sites) is either untargeted or contextually targeted, and can be discarded. Furthermore, any ad shown to >1 persona may be behaviorally targeted, but it cannot be a retarget, and is therefore filtered out⁴.

³<https://easylist-downloads.adblockplus.org/easylist.txt>

⁴Several of our personas have retailers in common, which we account for when filtering ads.

Figure 5 shows the average number of images remaining per persona after applying each filter. After applying all four filters, we are left with 31,850 ad images.

5.2 Identifying Targeted & Retargeted Ads

At this point, we do not know which of the ad images are retargets. Prior work has identified retargets by looking for specific URL parameters associated with them, however this technique is only able to identify a subset of retargets served by DoubleClick [43]. Since our goal is to be mechanism and ad exchange agnostic, we must use a more generalizable method to identify retargets.

Crowdsourcing. Given the large number of ads in our corpus, we decided to crowdsource labels from workers on Amazon Mechanical Turk (AMT). We constructed Human Intelligence Tasks (HITs) that ask workers to label 30 ads, 27 of which are unlabeled, and 3 of which are known to be retargeted ads and serve as controls (we manually identified 1,016 retargets from our corpus of 31,850 to serve as these controls).

Figure 6(a) shows a screenshot of our HIT. On the right is an ad image, and on the left we ask the worker two questions:

1. Does the image belong to one of the following categories (with “None of the above” being one option)?
2. Does the image say it came from one of the following websites (with “No” being one option)?

The purpose of question (1) is to isolate behavioral and retargeted ads from contextual and untargeted ads (*e.g.*, Figure 6(c), which was served to our *Music* persona). The list for question (1) is populated with the shopping categories associated with the persona that crawled the ad. For example, as shown in Figure 6(a), the category list includes “shopping_jewelry_diamonds” for ads shown to our *Diamond Jewelry* persona. In most cases, this list contains exactly one entry, although there are rare cases where up to 3 categories are in the list.

If the worker does not select “None” for question (1), then they are shown question (2). Question (2) is designed to separate retargets from behavioral targeted ads. The list of websites for question (2) is populated with the e-commerce sites visited by the persona that crawled the ad. For example, in Figure 6(a), the ad clearly says “Adiamor”, and one of the sites visited by the persona is adiamor.com; thus, this image is likely to be a retarget.

Quality Control. We apply four widely used techniques to maintain and validate the quality of our crowdsourced image labels [63, 29, 56]. *First*, we restrict our HITs to workers that have completed ≥ 50 HITs and have an approval rating $\geq 95\%$. *Second*, we restrict our HITs to workers living in the US, since our ads were collected

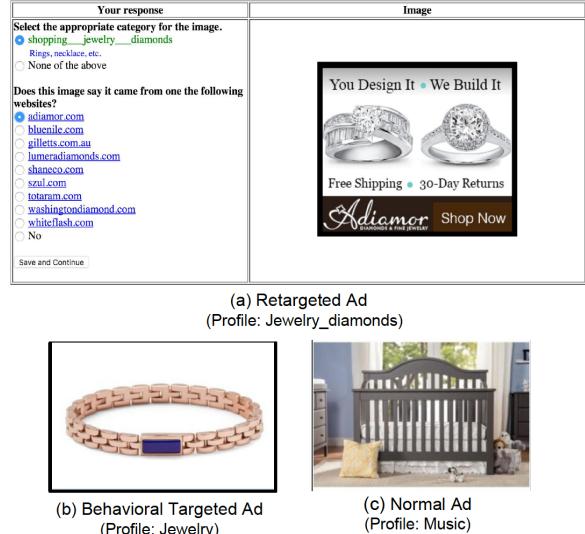


Figure 6: Screenshot of our AMT HIT, and examples of different types of ads.

from US websites. *Third*, we reject a HIT if the worker mislabels ≥ 2 of the control images (*i.e.*, known retargeted ads); this prevents workers from being able to simply answer “None” to all questions. We resubmitted rejected HITs for completion by another worker. Overall, the workers correctly labeled 87% of the control images. *Fourth* and finally, we obtain two labels on each unlabeled image by different workers. For 92.4% of images both labels match, so we accept them. We manually labeled the divergent images ourselves to break the tie.

Finding More Retargets. The workers from AMT successfully identified 1,359 retargeted ads. However, it is possible that they failed to identify some retargets, *i.e.*, there are false negatives. This may occur in cases like Figure 6(b): it is not clear if this ad was served as a behavioral target based on the persona’s interest in jewelry, or as a retarget for a specific jeweler.

To mitigate this issue, we manually examined all 7,563 images that were labeled as behavioral ads by the workers. In addition to the images themselves, we also looked at the inclusion chains for each image. In many cases, the URLs reveal that specific e-commerce sites visited by our personas hosted the images, indicating that the ads are retargets. For example, Figure 6(b) is actually part of a retargeted ad from fossil.com. Our manual analysis uncovered an additional 3,743 retargeted ads.

These results suggest that the number of false negatives from our crowdsourcing task could be dramatically reduced by showing the URLs associated with each ad image to the workers. However, note that adding this information to the HIT will change the dynamics of the

task: false negatives may go down but the effort (and therefore the cost) of each HIT will go up. This stems from the additional time it will take each worker to review the ad URLs for relevant keywords.

In § 6.2, we compare the datasets labeled by the workers and by the authors. Interestingly, although our dataset contains a greater *magnitude* of retargeted ads versus the worker’s dataset, it does not improve *diversity*, *i.e.*, the smaller dataset identifies 96% of the top 25 most frequent ad networks in the larger dataset. These networks are responsible for the vast majority of retargeted ads and inclusion chains in our dataset.

Final Results. Overall, we submitted 1,142 HITs to AMT. We paid \$0.18 per HIT, for a total of \$415. We did not collect any personal information from workers. In total, we and workers from AMT labeled 31,850 images, of which 7,563 are behavioral targeted ads and 5,102 are retargeted ads. These retargets advertise 281 distinct e-commerce websites (38% of all e-commerce sites).

5.3 Limitations

With any labeling task of this size and complexity, it is possible that there are false positives and negatives. Unfortunately, we cannot bound these quantities, since we do not have ground-truth information about known retargeted ad campaigns, nor is there a reliable mechanism to automatically detect retargets (*e.g.*, based on special URL parameters, *etc.*).

In practice, the effect of false positives is that we will erroneously classify pairs of ad exchanges as sharing information. We take measures to mitigate false positives by running a control crawl and removing images which appear in multiple personas (see § 5.1), but false positives can still occur. However, as we show in § 6, the results of our classifier are extremely consistent, suggesting that there are few false positives in our dataset.

False negatives have the opposite effect: we may miss pairs of ad exchanges that are sharing information. Fortunately, the practical impact of false negatives is low, since we only need to correctly identify a single retargeted ad to infer that a given pair of ad exchanges are sharing information.

6 Analysis

In this section, we use the 5,102 retargeted ads uncovered in § 5, coupled with their associated inclusion chains (see § 4.2), to analyze the information flows between ad exchanges. Specifically, we seek to answer two fundamental questions: *who* is sharing user data, and *how* does the sharing take place (*e.g.*, client-side via cookie matching, or server-side)?

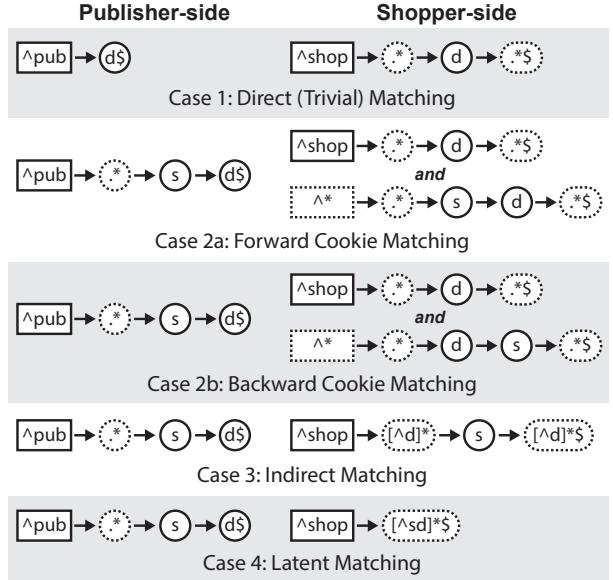


Figure 7: Regex-like rules we use to identify different types of ad exchange interactions. *shop* and *pub* refer to chains that begin at an e-commerce site or publisher, respectively. *d* is the DSP that serves a retarget; *s* is the predecessor to *d* in the publisher-side chain, and is most likely an SSP holding an auction. Dot star (.) matches any domains zero or more times.

We begin by *categorizing* all of the retargeted ads and their associated inclusion chains into one of four classes, which correspond to different mechanisms for sharing user data. Next, we examine specific pairs of ad exchanges that share data, and compare our detection approach to those used in prior work to identify cookie matching [43, 2, 54, 21]. We find that prior work may be missing 31% of collaborating exchanges. Finally, we construct a graph that captures ad exchanges and the relationships between them, and use it to reveal nuanced characteristics about the roles that different exchanges play in the ad ecosystem.

6.1 Information Flow Categorization

We begin our analysis by answering two basic questions: *for a given retargeted ad, was user information shared between ad exchanges, and if so, how?* To answer these questions, we categorize the 35,448 *publisher-side* inclusion chains corresponding to the 5,102 retargeted ads in our data. Note that 1) we observe some retargeted ads multiple times, resulting in multiple chains, and 2) the chains for a given unique ad may not be identical.

We place publisher-side chains into one of four categories, each of which corresponds to a specific information sharing mechanism (or lack thereof). To determine

the category of a given chain, we match it against carefully designed, regular expression-like rules. Figure 7 shows the pattern matching rules that we use to identify chains in each category. These rules are mutually exclusive, *i.e.*, a chain will match one or none of them.

Terminology. Before we explain each classification in detail, we first introduce shared terminology that will be used throughout this section. Each retargeted ad was served to our persona via a *publisher-side* chain. *pub* is the domain of the publisher at the root of the chain, while *d* is the domain at the end of the chain that served the ad. Typically, *d* is a DSP. If the retarget was served via an auction, then an SSP *s* must immediately precede *d* in the publisher-side chain.

Each retarget advertises a particular e-commerce site. *shop* is the domain of the e-commerce site corresponding to a particular retargeted ad. To categorize a given publisher-side chain, we must also consider the corresponding *shopper-side* chains rooted at *shop*.

6.1.1 Categorization Rules

Case 1: Direct Matches. The first chain type that we define are *direct matches*. Direct matches are the simplest type of chains that can be used to serve a retargeted ad. As shown in Figure 7, for us to categorize a publisher-side chain as a direct match, it must be exactly length two, with a direct resource inclusion request from *pub* to *d*. *d* receives any cookies they have stored on the persona inside this request, and thus it is trivial for *d* to identify our persona.

On the shopper-side, the only requirement is that *d* observed our persona browsing *shop*. If *d* does not observe our persona at *shop*, then *d* would not serve the persona a retargeted ad for *shop*. *d* is able to set a cookie on our persona, allowing *d* to re-identify the persona in future.

We refer to direct matching chains as “trivial” because it is obvious how *d* is able to track our persona and serve a retargeted ad for *shop*. Furthermore, in these cases no user information needs to be shared between ad exchanges, since there are no ad auctions being held on the publisher-side.

Case 2: Cookie Matching. The second chain type that we define are *cookie matches*. As the name implies, chains in this category correspond to instances where an auction is held on the publisher-side, and we observe direct resource inclusion requests between the SSP and DSP, implying that they are matching cookies.

As shown in Figure 7, for us to categorize a publisher-side chain as cookie matching, *s* and *d* must be adjacent at the end of the chain. On the shopper-side, *d* must observe the persona at *shop*. Lastly, we must observe a request from *s* to *d* or from *d* to *s* in some chain before

the retargeted ad is served. These requests capture the moment when the two ad exchanges match their cookies. Note that *s* → *d* or *d* → *s* can occur in a publisher- or shopper-side chain; in practice, it often occurs in a chain rooted at *shop*, thus fulfilling both requirements at once.

For the purposes of our analysis, we distinguish between *forward* (*s* → *d*) and *backward* (*d* → *s*) cookie matches. Figure 2 shows an example of a forward cookie match. As we will see, many pairs of ad exchanges engage in both forward and backward matching to maximize their opportunities for data sharing. To our knowledge, no prior work examines the distinction between forward and backward cookie matching.

Case 3: Indirect Matching. The third chain type we define are *indirect matches*. Indirect matching occurs when an SSP sends meta-data about a user to a DSP, to help them determine if they should bid on an impression. With respect to retargeted ads, the SSP tells the DSPs about the browsing history of the user, thus enabling the DSPs to serve retargets for specific retailers, even if the DSP never directly observed the user browsing the retailer (hence the name, *indirect*). Note that no cookie matching is necessary in this case for DSPs to serve retargeted ads.

As shown in Figure 7, the crucial difference between cookie matching chains and indirect chains is that *d* *never* observes our persona at *shop*; only *s* observes our persona at *shop*. Thus, by inductive reasoning, we must conclude that *s* shares information about our persona with *d*, otherwise *d* would never serve the persona a retarget for *shop*.

Case 4: Latent Matching. The fourth and final chain type that we define are *latent matches*. As shown in Figure 7, the defining characteristic of latent chains is that neither *s* nor *d* observe our persona at *shop*. This begs the question: how do *s* and *d* know to serve a retargeted ad for *shop* if they never observe our persona at *shop*? The most reasonable explanation is that some other ad exchange *x* that is present in the shopper-side chains shares this information with *d* behind-the-scenes.

We hypothesize that the simplest way for ad exchanges to implement latent matching is by having *x* and *d* share the same unique identifiers for users. Although *x* and *d* are different domains, and are thus prevented by the SOP from reading each others’ cookies, both ad exchanges may use the same deterministic algorithm for generating user IDs (*e.g.*, by relying on IP addresses or browser fingerprints). However, as we will show, these synchronized identifiers are not necessarily visible from the client-side (*i.e.*, the values of cookies set by *x* and *d* may be obfuscated), which prevents trivial identification of latent cookie matching.

Type	Unclustered Chains	%	Clustered Chains	%
Direct	1770	5%	8449	24%
Forward Cookie Match	24575	69%	25873	73%
Backward Cookie Match	19388	55%	24994	70%
Indirect Match	2492	7%	178	1%
Latent Match	5362	15%	343	1%
No Match	775	2%	183	1%

Table 1: Results of categorizing publisher-side chains, before and after clustering domains.

Note: Although we do not expect to see cases 3 and 4, they can still occur. We explain in § 6.1.2 that indirect and latent matching is mostly performed by domains belonging to the same company. The remaining few instances of these cases are probably mislabeled behaviorally targeted ads.

6.1.2 Categorization Results

We applied the rules in Figure 7 to all 35,448 publisher-side chains in our dataset twice. First, we categorized the raw, unmodified chains; then we *clustered* domains that belong to the same companies, and categorized the chains again. For example, Google owns youtube.com, doubleclick.com, and 2mdn.net; in the clustered experiments, we replace all instances of these domains with google.com. Appendix A.1 lists all clustered domains.

Table 1 presents the results of our categorization. The first thing we observe is that cookie matching is the most frequent classification by a large margin. This conforms to our expectations, given that RTB is widespread in today’s ad ecosystem, and major exchanges like DoubleClick support it [17]. Note that, for a given (s, d) pair in a publisher-side chain, we may observe $s \rightarrow d$ and $d \rightarrow s$ requests in our data, *i.e.*, the pair engages in forward and backward cookie matching. This explains why the percentages in Table 1 do not add up to 100%.

The next interesting feature that we observe in Table 1 is that indirect and latent matches are relatively rare (7% and 15%, respectively). Again, this is expected, since these types of matching are more exotic and require a greater degree of collaboration between ad exchanges to implement. Furthermore, the percentage of indirect and latent matches drops to 1% when we cluster domains. To understand why this occurs, consider the following real-world example chains:

Publisher-side: $pub \rightarrow rubicon \rightarrow googlesyndication$

Shopper-side: $shop \rightarrow doubleclick$

According to the rules in Figure 7, this appears to be a latent match, since Rubicon and Google Syndication do not observe our persona on the shopper-side. However, after clustering the Google domains, this will be clas-

sified as cookie matching (assuming that there exists at least one other request from Rubicon to Google).

The above example is extremely common in our dataset: 731 indirect chains become cookie matching chains after we cluster the Google domains *alone*. Importantly, this finding provides strong evidence that Google does in fact use latent matching to share user tracking data between its various domains. Although this is allowed in Google’s terms of service as of 2014 [26], our results provide direct evidence of this data sharing with respect to serving targeted ads. In the vast majority of these cases, Google Syndication is the DSP, suggesting that on the server-side, it ingests tracking data and user identifiers from all other Google services (*e.g.*, DoubleClick and Google Tag Manager).

Of the remaining 1% of chains that are still classified as indirect or latent after clustering, the majority appear to be false positives. In most of these cases, we observe s and d doing cookie matching in other instances, and it seems unlikely that s and d would also utilize indirect and latent mechanisms. These ads are probably mislabeled behaviorally targeted ads.

The final takeaway from Table 1 is that the number of uncategorized chains that do not match any of our rules is extremely low (1-2%). These publisher-side chains are likely to be false positives, *i.e.*, ads that are not actually retargeted. These results suggest that our image labeling approach is very robust, since the vast majority of chains are properly classified as direct or cookie matches.

6.2 Cookie Matching

The results from the previous section confirm that cookie matching is ubiquitous on today’s Web, and that this information sharing fuels highly targeted advertisements. Furthermore, our classification results demonstrate that we can detect cookie matching without relying on semantic information about cookie matching mechanisms.

In this section, we take a closer look at the pairs of ad exchanges that we observe matching cookies. We seek to answer two questions: *first*, which pairs match most frequently, and what is the directionality of these relationships? *Second*, what fraction of cookie matching relationships will be missed by the heuristic detection approaches used by prior work [43, 2, 54, 21]?

Who Is Cookie Matching? Table 2 shows the top 25 most frequent pairs of domains that we observe matching cookies. The arrows indicate the direction of matching (forward, backward, or both). “Ads” is the number of unique retargets served by the pair, while “Chains” is the total number of associated publisher-side chains. We present both quantities as observed in our complete dataset (containing 5,102 retargets), as well as the subset

Participant 1	Participant 2	All Data		AMT Only		Heuristics
		Chains	Ads	Chains	Ads	
criteo	↔ googlesyndication	9090	1887	1629	370	↔: US
criteo	↔ doubleclick	3610	1144	770	220	→: E, US ←: DC, US
criteo	↔ adnxs	3263	1066	511	174	↔: E, US
criteo	↔ googleadservices	2184	1030	448	214	→: E, US ←: US
criteo	↔ rubiconproject	1586	749	240	113	↔: E, US
criteo	↔ servedbyopenx	707	460	111	71	↔: US
mythings	↔ mythingsmedia	478	52	53	1	→: E, US ←: US
criteo	↔ pubmatic	363	246	64	37	→: E, US ←: US
doubleclick	↔ steelhousemedia	362	27	151	16	→: US ←: E, US
mathtag	↔ mediaforge	360	124	63	13	↔: E, US
netmng	↔ scene7	267	162	45	32	→: E ←: -
criteo	↔ casalemedia	200	119	54	31	→: E, US ←: US
doubleclick	↔ googlesyndication	195	81	101	62	↔: US
criteo	↔ clickfuse	126	99	14	13	↔: US
criteo	↔ bidswitch	112	78	25	15	→: E, US ←: US
googlesyndication	↔ adsrvr	107	29	102	24	↔: US
rubiconproject	↔ steelhousemedia	86	30	43	19	↔: E
amazon-adsystem	↔ ssl-images-amazon	98	33	33	7	-
googlesyndication	↔ steelhousemedia	47	22	36	16	-
adtechus	→ adacado	36	18	36	18	-
googlesyndication	↔ 2mdn	40	19	39	18	→: US ←: -
atwola	→ adacado	32	6	28	5	-
adroll	↔ adnxs	31	8	26	7	-
googlesyndication	↔ adlegend	31	22	29	20	-
adnxs	↔ esm1	46	1	0	0	→: US ←: -

Table 2: Top 25 cookie matching partners in our dataset. The arrow signifies whether we observe forward matches (\rightarrow), backward matches (\leftarrow), or both (\leftrightarrow). The heuristics for detecting cookie matching are: *DC* (match using DoubleClick URL parameters), *E* (string match for exact cookie values), *US* (URLs that include parameters like “usersync”), and - (no identifiable mechanisms). Note that the HTTP request formats used for forward and backward matches between a given pair of exchanges may vary.

that was identified solely by the AMT workers (containing 1,359 retargets).

We observe that cookie matching frequency is heavily skewed towards several heavy-hitters. In aggregate, Google’s domains are most common, which makes sense given that Google is the largest ad exchange on the Web today. The second most common is Criteo; this result also makes sense, given that Criteo specializes in retargeted advertising [15]. These observations remain broadly true across the AMT and complete datasets: although the relative proportion of ads and chains from less-frequent exchange pairs differs somewhat between the two datasets, the heavy-hitters do not change. Furthermore, we also see that the vast majority of exchange pairs are identified in both datasets.

Interestingly, we observe a great deal of heterogeneity with respect to the directionality of cookie matching. Some boutique exchanges, like Adacado, only ingest cookies from other exchanges. Others, like Criteo, are omnivorous, sending or receiving data from any and all willing partners. These results suggest that some participants are more wary about releasing their user identifiers to other exchanges.

Comparison to Prior Work. We observe many of the same participants matching cookies as prior work, in-

cluding DoubleClick, Rubicon, AppNexus, OpenX, MediaMath, and myThings [2, 54, 21]. Prior work identifies some additional ad exchanges that we do not (*e.g.*, Turn); this is due to our exclusive focus on participants involved in retargeted advertising.

However, we also observe participants (*e.g.*, Adacado and AdRoll) that prior work does not. This may be because prior work identifies cookie matching using heuristics to pick out specific features in HTTP requests [43, 2, 54, 21]. In contrast, our categorization approach is content and mechanism agnostic.

To investigate the efficacy of heuristic detection methods, we applied three of them to our dataset. Specifically, for each pair (s, d) of exchanges that we categorize as cookie matching, we apply the following tests to the HTTP headers of requests between s and d or vice-versa:

1. We look for specific keys that are known to be used by DoubleClick and other Google domains for cookie matching (*e.g.*, “google_nid” [54]).
2. We look for cases where unique cookie values set by one participant are included in requests sent to the other participant⁵.

⁵To reduce false positives, we only consider cookie values that have length > 10 and < 100 .

	Domain	Degree			Position p in Chains (%)			# of Shopper Websites	# of Ads
		In	Out	In/Out Ratio	p_2	p_{n-1}	p_n		
DSPs	criteo	35	6	5.83	9.28	0.00	68.8	248	3,335
	mediaplex	8	2	4.00	0.00	85.7	0.07	20	14
	tellapart	6	1	6.00	25.0	100.0	0.18	33	9
	mathtag	12	6	2.00	0.00	90.9	0.06	314	2
	mythingsmedia	1	0	-	0.00	0.00	1.41	1	59
	steelhousemedia	8	0	-	0.00	0.00	16.8	40	89
SSPs	mediaforge	5	0	-	0.00	0.00	1.28	29	143
	pubmatic	5	9	0.56	3.17	74.2	0.01	362	4
	rubiconproject	19	22	0.86	23.5	62.8	0.01	394	3
	adnx	18	20	0.90	94.2	91.9	0.16	476	12
	casalemedia	9	10	0.90	1.30	90.0	0.00	298	0
	atwola	4	19	0.21	84.6	18.2	0.01	62	2
AOL	advertising	4	4	1.00	0.00	75.0	0.10	337	17
	adtechus	17	16	1.06	1.58	27.3	0.09	328	15
	servedbyopenx	6	11	0.55	7.2	83.8	0.00	2	0
OpenX	openx	10	9	1.11	0.95	9.83	0.00	390	0
	openxenterprise	4	4	1.00	40.0	20.0	0.00	1	0
	googletagservices	44	2	22.00	93.7	0.00	0.00	65	0
Google	googleleadservices	4	17	0.24	2.94	33.5	0.00	485	0
	2mdn	3	1	3.00	0.00	0.00	1.35	62	125
	googlesyndication	90	35	2.57	70.1	62.7	19.8	84	638
	doubleclick	38	36	1.06	38.8	63.1	0.22	675	19

Table 3: Overall statistics about the connectivity, position, and frequency of ad domains in our dataset.

- We look for keys with revealing names like “user-sync” that frequently appear in requests between participants in our data.

As shown in the “Heuristics” column in Table 2, in the majority of cases, heuristics are able to identify cookie matching between the participants. Interestingly, we observe that the mechanisms used by some pairs (*e.g.*, Criteo and DoubleClick) change depending on the directionality of the cookie match, revealing that the two sides have different cookie matching APIs.

However, for 31% of our cookie matching partners, the heuristics are unable to detect signs of cookie matching. We hypothesize that this is due to obfuscation techniques employed by specific ad exchanges. In total, there are 4.1% cookie matching chains that would be completely missed by heuristic tests. This finding highlights the limitations of prior work, and bolsters the case for our mechanism-agnostic classification methodology.

6.3 The Retargeting Ecosystem

In this last section, we take a step back and examine the broader ecosystem for retargeted ads that is revealed by our dataset. To facilitate this analysis, we construct a graph by taking the union of all of our publisher-side chains. In this graph, each node is a domain (either a publisher or an ad exchange), and edges correspond to resource inclusion relationships between the domains. Our graph formulation differs from prior work in that edges denote actual information flows, as opposed to simple co-occurrences of trackers on a given domain [25].

Table 3 presents statistics on the top ad-related domains in our dataset. The “Degree” column shows the in- and out-degree of nodes, while “Position” looks at the relative location of nodes within chains. p_2 is the second position in the chain, corresponding to the first ad network after the publisher; p_n is the DSP that serves the retarget in a chain of length n ; p_{n-1} is the second to last position, corresponding to the final SSP before the DSP. Note that a domain may appear in a chain multiple times, so the sum of the p_i percentages may be $>100\%$. The last two columns count the number of unique e-commerce sites that embed resources from a given domain, and the unique number of ads served by the domain.

Based on the data in Table 3, we can roughly cluster the ad domains into two groups, corresponding to SSPs and DSPs. DSPs have low or zero out-degree since they often appear at position p_n , *i.e.*, they serve an ad and terminate the chain. Criteo is the largest source of retargeted ads in our dataset by an order of magnitude. This is not surprising, given that Criteo was identified as the largest retargeter in the US and UK in 2014 [15].

In contrast, SSPs tend to have in/out degree ratios closer to 1, since they facilitate the exchange of ads between multiple publishers, DSPs, and even other SSPs. Some SSPs, like Atwola, work more closely with publishers and thus appear more frequently at p_2 , while others, like Mathtag, cater to other SSPs and thus appear almost exclusively at p_{n-1} . Most of the SSPs we observe also function as DSPs (*i.e.*, they serve some retargeted ads), but there are “pure” SSPs like Casale Media and OpenX that do not serve ads. Lastly, Table 3 reveals that

SSPs tend to do more user tracking than DSPs, by getting embedded in more e-commerce sites (with Criteo being the notable exception).

Google is an interesting case study because its different domains have clearly delineated purposes. `googletagservices` is Google’s in-house SSP, which funnels impressions directly from publishers to Google’s DSPs: `2mdn`, `googlesyndication`, and `doubleclick`. In contrast, `googleadservices` is also an SSP, but it holds auctions with third-party participants (*e.g.*, Criteo). `googlesyndication` and `doubleclick` function as both SSPs and DSPs, sometimes holding auctions, and sometimes winning auctions held by others to serve ads. Google Syndication is the second most frequent source of retargeted ads in our dataset behind Criteo.

7 Concluding Discussion

In this study, we develop a novel, principled methodology for detecting flows of tracking information between ad exchanges. The key insight behind our approach is that we re-purpose retargeted ads as a detection mechanism, since their presence reveals information flows between ad exchanges. Our methodology is content-agnostic, and thus we are able to identify flows even if they occur on the server-side. This is a significant improvement over prior work, which relies on heuristics to detect cookie matching [2, 54, 21]. As we show in § 6, these heuristics fail to detect 31% of matching pairs today, and they are likely to fail more in the future as ad networks adopt content obfuscation techniques.

Implications for Users. Ultimately, our goal is not just to measure information flows between ad exchanges, but to facilitate the development of systems that balance user privacy against the revenue needs of publishers.

Currently, users are faced with unsatisfactory choices when deciding if and how to block ads and tracking. Whitelisting approaches like NoScript are effective at protecting privacy, but are too complicated for most users, and deprive publishers of revenue. Blocking third-party cookies is ineffective against first-party trackers (*e.g.*, Facebook). AdBlockPlus’ controversial “Acceptable Ads” program is poorly governed and leaves users vulnerable to unscrupulous ad networks [62]. DNT is DOA [8]. Although researchers have proposed privacy preserving ad exchanges, these systems have yet to see widespread adoption [22, 28, 7].

We believe that data about information flows between ad exchanges potentially opens up a new middle ground in ad blocking. One possibility is to develop an automated system that uses the methodology developed in this paper to continuously crawl ads, identify cookie matching flows, and construct rules that match these

flows. Users could then install a browser extension that blocks flows matching these rules. The advantage of this extension is that it would offer improved privacy protection relative to existing systems (*e.g.*, Ghostery and Disconnect), while also allowing advertising (as opposed to traditional ad blockers). However, the open challenge with this system design would be making it cost effective, since it would still rely on crowdsourced labor.

Another possibility is using our data as ground-truth for a sophisticated blocker that relies on client-side Information Flow Control (IFC). There exist many promising, lightweight approaches to implementing JavaScript IFC in the browser [30, 10, 59, 31]. However, IFC alone is not enough to block cookie matching flows: as we have shown, ad networks obfuscate data, making it impossible to separate benign from “leaky” flows in general. Instead, we can use information gathered using our methodology as ground-truth to mark data in specific incoming flows, and rely on IFC to enforce restrictions that prevent outgoing flows from containing the marked data.

Acknowledgements

We thank our shepherd, Nektarios Leontiadis, and the anonymous reviewers for their helpful comments. This research was supported in part by NSF grants CNS-1319019 and CHS-1408345. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

- [1] Real-time bidding protocol, February 2016. <https://developers.google.com/ad-exchange/rtb/cookie-guide>.
- [2] ACAR, G., EUBANK, C., ENGLEHARDT, S., JUAREZ, M., NARAYANAN, A., AND DIAZ, C. The web never forgets: Persistent tracking mechanisms in the wild. In *Proc. of CCS* (2014).
- [3] ACAR, G., JUAREZ, M., NIKIFORAKIS, N., DIAZ, C., GÜRSES, S., PIJSESENS, F., AND PRENEEL, B. Fpdetective: Dusting the web for fingerprinters. In *Proc. of CCS* (2013).
- [4] AGARWAL, L., SHRIVASTAVA, N., JAISWAL, S., AND PANJWANI, S. Do not embarrass: Re-examining user concerns for online tracking and advertising.
- [5] ARSHAD, S., KHARRAZ, A., AND ROBERTSON, W. Include me out: In-browser detection of malicious third-party content inclusions. In *Proc. of Intl. Conf. on Financial Cryptography* (2016).
- [6] AYENSON, M., WAMBACH, D. J., SOLTANI, A., GOOD, N., AND HOOFNAGLE, C. J. Flash cookies and privacy ii: Now with html5 and etag respawning. Available at SSRN 1898390 (2011).
- [7] BACKES, M., KATE, A., MAFFEI, M., AND PECINA, K. Obliviad: Provably secure and practical online behavioral advertising. In *Proc. of IEEE Symposium on Security and Privacy* (2012).

- [8] BALEBAKO, R., LEON, P. G., SHAY, R., UR, B., WANG, Y., AND CRANOR, L. F. Measuring the effectiveness of privacy tools for limiting behavioral advertising. In *Proc. of W2SP* (2012).
- [9] BARFORD, P., CANADI, I., KRUSHEVSKAJA, D., MA, Q., AND MUTHUKRISHNAN, S. Adscape: Harvesting and analyzing online display ads. In *Proc. of WWW* (2014).
- [10] BICHHAWAT, A., RAJANI, V., GARG, D., AND HAMMER, C. Information flow control in webkit's javascript bytecode. In *Proc. of Principles of Security and Trust* (2014).
- [11] CAHN, A., ALFELD, S., BARFORD, P., AND MUTHUKRISHNAN, S. An empirical study of web cookies. In *Proc. of WWW* (2016).
- [12] CARRASCOSA, J. M., MIKIANI, J., CUEVAS, R., ERRAMILLI, V., AND LAOUTARIS, N. I always feel like somebody's watching me: Measuring online behavioural advertising. In *Proc. of ACM CoNEXT* (2015).
- [13] CASTELLUCCIA, C., KAAFAR, M.-A., AND TRAN, M.-D. Betrayed by your ads!: Reconstructing user profiles from targeted ads. In *Proc. of PETS* (2012).
- [14] CHANCHARY, F., AND CHIASSON, S. User perceptions of sharing, advertising, and tracking.
- [15] Criteo ranking by Econsultancy. <http://www.criteo.com/resources/e-consultancy-display-retargeting-buyers-guide/>.
- [16] DATTA, A., TSCHANTZ, M. C., AND DATTA, A. Automated experiments on ad privacy settings: A tale of opacity, choice, and discrimination. In *Proc. of PETS* (2015).
- [17] Double Click RTB explained. <https://developers.google.com/ad-exchange/rtb/>.
- [18] ECKERSLEY, P. How unique is your web browser? In *Proc. of PETS* (2010).
- [19] ENGLEHARDT, S., REISMAN, D., EUBANK, C., ZIMMERMAN, P., MAYER, J., NARAYANAN, A., AND FELTN, E. W. Cookies that give you away: The surveillance implications of web tracking. In *Proc. of WWW* (2015).
- [20] FALAHRASTEGAR, M., HADDADI, H., UHLIG, S., AND MORTIER, R. The rise of panopticons: Examining region-specific third-party web tracking. In *Proc. of Traffic Monitoring and Analysis* (2014).
- [21] FALAHRASTEGAR, M., HADDADI, H., UHLIG, S., AND MORTIER, R. Tracking personal identifiers across the web. In *Proc. of PAM* (2016).
- [22] FREDRIKSON, M., AND LIVSHITS, B. Repriv: Re-imaging content personalization and in-browser privacy. In *Proc. of IEEE Symposium on Security and Privacy* (2011).
- [23] GHOSH, A., MAHDIAN, M., McAFFEE, P., AND VASSILVITSKII, S. To match or not to match: Economics of cookie matching in online advertising. In *Proc. of EC* (2012).
- [24] GILL, P., ERRAMILLI, V., CHAINTREAU, A., KRISHNAMURTHY, B., PAPAGIANNAKI, K., AND RODRIGUEZ, P. Follow the money: Understanding economics of online aggregation and advertising. In *Proc. of IMC* (2013).
- [25] GOMER, R., RODRIGUES, E. M., MILIC-FRAYLING, N., AND SCHRAEFEL, M. C. Network analysis of third party tracking: User exposure to tracking cookies through search. In *Proc. of IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)* (2013).
- [26] GOODALE, G. Privacy concerns? what google now says it can do with your data. Christian Science Monitor, April 2014. <http://www.csmonitor.com/USA/2014/0416/Privacy-concerns-What-Google-now-says-it-can-do-with-your-data-video>.
- [27] GUHA, S., CHENG, B., AND FRANCIS, P. Challenges in measuring online advertising systems. In *Proc. of IMC* (2010).
- [28] GUHA, S., CHENG, B., AND FRANCIS, P. Privad: Practical privacy in online advertising. In *Proc. of NSDI* (2011).
- [29] HANNAK, A., SAPIEŻYŃSKI, P., KAKHKI, A. M., KRISHNAMURTHY, B., LAZER, D., MISLOVE, A., AND WILSON, C. Measuring Personalization of Web Search. In *Proc. of WWW* (2013).
- [30] HEDIN, D., BIRGISSON, A., BELLO, L., AND SABELFELD, A. JSFlow: Tracking Information Flow in JavaScript and Its APIs. In *Proc. of Symposium on Applied Computing* (2014).
- [31] HEULE, S., STEFAN, D., YANG, E. Z., MITCHELL, J. C., AND RUSSO, A. IFC inside: Retrofitting languages with dynamic information flow control. In *Proc. of Principles of Security and Trust* (2015).
- [32] HOOFNAGLE, C. J., AND URBAN, J. M. Alan westin's privacy homo economicus. *49 Wake Forest Law Review* 261 (2014).
- [33] HOWELL, D. How to protect your privacy and remove data from online services. Tech Radar, January 2015. <http://www.techradar.com/news/internet/how-to-protect-your-privacy-and-remove-data-from-online-services-1291515>.
- [34] KAMKAR, S. Evercookie - virtually irrevocable persistent cookies., September 2010. <http://samy.pl/evercookie/>.
- [35] KOHNO, T., BRODIO, A., AND CLAFFY, K. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing* 2, 2 (2005), 93–108.
- [36] KRISHNAMURTHY, B., NARYSHKIN, K., AND WILLS, C. Privacy diffusion on the web: A longitudinal perspective. In *Proc. of WWW* (2009).
- [37] KRISHNAMURTHY, B., AND WILLS, C. Privacy leakage vs. protection measures: the growing disconnect. In *Proc. of W2SP* (2011).
- [38] KRISHNAMURTHY, B., AND WILLS, C. E. Generating a privacy footprint on the internet. In *Proc. of IMC* (2006).
- [39] LÉCUYER, M., DU COFFE, G., LAN, F., PAPANCEA, A., PETSIOS, T., SPAHN, R., CHAINTREAU, A., AND GEAMBASU, R. Xray: Enhancing the web's transparency with differential correlation. In *Proc. of USENIX Security Symposium* (2014).
- [40] LÉCUYER, M., SPAHN, R., SPILIOPOLOUS, Y., CHAINTREAU, A., GEAMBASU, R., AND HSU, D. Sunlight: Fine-grained targeting detection at scale with statistical confidence. In *Proc. of CCS* (2015).
- [41] LEON, P. G., UR, B., WANG, Y., SLEEPER, M., BALEBAKO, R., SHAY, R., BAUER, L., CHRISTODORESCU, M., AND CRANOR, L. F. What matters to users?: Factors that affect users' willingness to share information with online advertisers.
- [42] LI, T.-C., HANG, H., FALOUTSOS, M., AND EFSTATHOPOULOS, P. Trackadvisor: Taking back browsing privacy from third-party trackers. In *Proc. of PAM* (2015).
- [43] LIU, B., SHETH, A., WEINSBERG, U., CHANDRASHEKAR, J., AND GOVINDAN, R. Adreveal: Improving transparency into online targeted advertising. In *Proc. of HotNets* (2013).
- [44] MALHEIROS, M., JENNITT, C., PATEL, S., BROSTOFF, S., AND SASSE, M. A. Too close for comfort: A study of the effectiveness and acceptability of rich-media personalized advertising. In *Proc. of CHI* (2012).
- [45] MAYER, J. R., AND MITCHELL, J. C. Third-party web tracking: Policy and technology. In *Proc. of IEEE Symposium on Security and Privacy* (2012).

- [46] McDONALD, A. M., AND CRANOR, L. F. Americans' attitudes about internet behavioral advertising practices. In *Proc. of WPES* (2010).
- [47] McDONALD, A. M., AND CRANOR, L. F. A survey of the use of adobe flash local shared objects to respawn http cookies. *ISJLP* 7, 639 (2011).
- [48] MOWERY, K., BOGENREIF, D., YILEK, S., AND SHACHAM, H. Fingerprinting information in JavaScript implementations. In *Proc. of W2SP* (2011).
- [49] MOWERY, K., AND SHACHAM, H. Pixel perfect: Fingerprinting canvas in html5. In *Proc. of W2SP* (2012).
- [50] MULAZZANI, M., RESCHL, P., HUBER, M., LEITHNER, M., SCHRITTWIESER, S., AND WEIPPL, E. Fast and reliable browser identification with JavaScript engine fingerprinting. In *Proc. of W2SP* (2013).
- [51] NIKIFORAKIS, N., JOOSEN, W., AND LIVSHITS, B. Privacicator: Deceiving fingerprinters with little white lies. In *Proc. of WWW* (2015).
- [52] NIKIFORAKIS, N., KAPRAVELOS, A., JOOSEN, W., KRUEGEL, C., PISSSENS, F., AND VIGNA, G. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proc. of IEEE Symposium on Security and Privacy* (2013).
- [53] OLEJNIK, L., CASTELLUCCIA, C., AND JANC, A. Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns. In *Proc. of HotPETs* (2012).
- [54] OLEJNIK, L., MINH-DUNG, T., AND CASTELLUCCIA, C. Selling off privacy at auction. In *Proc. of NDSS* (2014).
- [55] ROESNER, F., KOHNO, T., AND WETHERALL, D. Detecting and defending against third-party tracking on the web. In *Proc. of NSDI* (2012).
- [56] SOELLER, G., KARAHALIOS, K., SANDVIG, C., AND WILSON, C. Mapwatch: Detecting and monitoring international border personalization on online maps. In *Proc. of WWW* (2016).
- [57] SOLTANI, A., CANTY, S., MAYO, Q., THOMAS, L., AND HOOFNAGLE, C. J. Flash cookies and privacy. In *AAAI Spring Symposium: Intelligent Information Privacy Management* (2010).
- [58] SPECTOR, L. Online privacy tips: 3 ways to control your digital footprint. PC World, January 2016. <http://www.pcworld.com/article/3020163/internet/online-privacy-tips-3-ways-to-control-your-digital-footprint.html>.
- [59] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIÈRES, D. Protecting users by confining JavaScript with COWL. In *Proc. of OSDI* (2014).
- [60] UR, B., LEON, P. G., CRANOR, L. F., SHAY, R., AND WANG, Y. Smart, useful, scary, creepy: Perceptions of online behavioral advertising.
- [61] VALLINA-RODRIGUEZ, N., SHAH, J., FINAMORE, A., GRUNENBERGER, Y., PAPAGIANNAKI, K., HADDADI, H., AND CROWCROFT, J. Breaking for commercials: Characterizing mobile advertising. In *Proc. of IMC* (2012).
- [62] WALLS, R. J., KILMER, E. D., LAGEMAN, N., AND MC DANIEL, P. D. Measuring the impact and perception of acceptable advertisements. In *Proc. of IMC* (2015).
- [63] WANG, G., MOHANLAL, M., WILSON, C., WANG, X., METZGER, M., ZHENG, H., AND ZHAO, B. Y. Social turing tests: Crowdsourcing sybil detection. In *Proc. of NDSS* (2013).
- [64] WILLS, C. E., AND TATAR, C. Understanding what they do with what they know. In *Proc. of WPES* (2012).
- [65] WOLPIN, S. International privacy day: Protect your digital footprint. The Huffington Post, January 2015. http://www.huffingtonpost.com/stewart-wolpin/international-privacy-day_b_6551012.html.
- [66] ZARRAS, A., KAPRAVELOS, A., STRINGHINI, G., HOLZ, T., KRUEGEL, C., AND VIGNA, G. The dark alleys of madison avenue: Understanding malicious advertisements. In *Proc. of IMC* (2014).

A Appendix

A.1 Clustered Domains

We clustered the following domains together when classifying publisher-side chains in § 6.1.2.

Google: google-analytics, googleapis, google, doubleclick, gstatic, googlesyndication, googleusercontent, googleleadservices, googletagmanager, googletagservices, googlecommerce, youtube, yimg, youtube-mp3, googlevideo, 2mdn

OpenX: openxenterprise, openx, servedbyopenx

Affinity: affinitymatrix, affinity

Ebay: ebay, ebaystatic

Yahoo: yahoo, yimg

Mythings: mythingsmedia, mythings

Amazon: cloudfront, amazonaws, amazon-adsystem, images-amazon

Tellapart: tellapart, tellaparts

Virtual U: Defeating Face Liveness Detection by Building Virtual Models From Your Public Photos

Yi Xu, True Price, Jan-Michael Frahm, Fabian Monroe

Department of Computer Science, University of North Carolina at Chapel Hill

{yix, jtprice, jmf, fabian}@cs.unc.edu

Abstract

In this paper, we introduce a novel approach to bypass modern face authentication systems. More specifically, by leveraging a handful of pictures of the target user taken from social media, we show how to create realistic, textured, 3D facial models that undermine the security of widely used face authentication solutions. Our framework makes use of virtual reality (VR) systems, incorporating along the way the ability to perform animations (*e.g.*, raising an eyebrow or smiling) of the facial model, in order to trick liveness detectors into believing that the 3D model is a real human face. The synthetic face of the user is displayed on the screen of the VR device, and as the device rotates and translates in the real world, the 3D face moves accordingly. To an observing face authentication system, the depth and motion cues of the display match what would be expected for a human face.

We argue that such VR-based spoofing attacks constitute a fundamentally new class of attacks that point to a serious weaknesses in camera-based authentication systems: Unless they incorporate other sources of verifiable data, systems relying on color image data and camera motion are prone to attacks via virtual realism. To demonstrate the practical nature of this threat, we conduct thorough experiments using an end-to-end implementation of our approach and show how it undermines the security of several face authentication solutions that include both motion-based and liveness detectors.

1 Introduction

Over the past few years, face authentication systems have become increasingly popular as an enhanced security feature in both mobile devices and desktop computers. As the underlying computer vision algorithms have matured, many application designers and nascent specialist vendors have jumped in and started to offer solutions for mobile devices with varying degrees of security and usability. Other more well-known players, like Apple and

Google, are poised to enter the market with their own solutions, having already acquired several facial recognition software companies¹. While the market is segmented based on the type of technology offered (*e.g.*, 2D facial recognition, 3D recognition, and facial analytics/face biometric authentication), Gartner research estimates that the overall market will grow to over \$6.5 billion in 2018 (compared to roughly \$2 billion today) [13].

With this push to market, improving the accuracy of face recognition technologies remains an active area of research in academia and industry. Google’s FaceNet system, which achieved near-perfect accuracy on the Labeled Faces in the Wild dataset [47], exemplifies one such effort. Additionally, recent advances with deep learning algorithms [38, 53] show much promise in strengthening the robustness of the face identification and authentication techniques used today. Indeed, state-of-the-art face identification systems can now outperform their human counterparts [36], and this high accuracy is one of the driving factors behind the increased use of face recognition systems.

However, even given the high accuracy of modern face recognition technologies, their application in face authentication systems has left much to be desired. For instance, at the Black Hat security conference in 2009, Duc and Minh [10] demonstrated the weaknesses of popular face authentication systems from commodity vendors like Lenovo, Asus, and Toshiba. Amusingly, Duc and Minh [10] were able to reliably bypass face-locked computers simply by presenting the software with photographs and fake pictures of faces. Essentially, the security of these systems rested solely on the problem of face detection, rather than face authentication. This widely publicized event led to subsequent integration of more robust face authentication protocols. One prominent example is Android OS, which augmented its face authen-

¹See, for example, “Apple Acquires Face Recognition, Expression Analysis firm, Emotient”, TechTimes, Jan, 2016; “Google Acquires Facial Recognition Software Company PittPar,” WSJ, 2011.

tication approach in 2012 to require users to blink while authenticating (*i.e.*, as a countermeasure to still-image spoofing attacks). Unfortunately, this approach was also shown to provide little protection, and can be easily bypassed by presenting the system with two alternating images — one with the user’s eyes open, and one with her eyes closed.² These attacks underscore the fact that face authentication systems require robust security features beyond mere recognition in order to foil spoofing attacks.

Loosely speaking, three types of such spoofing attacks have been used in the past, to varying degrees of success: (*i*) still-image-based spoofing, (*ii*) video-based spoofing, and (*iii*) 3D-mask-based spoofing. As the name suggests, still-image-based spoofing attacks present one or more still images of the user to the authentication camera; each image is either printed on paper or shown with a digitized display. Video-based spoofing, on the other hand, presents a pre-recorded video of the victim’s moving face in an attempt to trick the system into falsely recognizing motion as an indication of liveness. The 3D-mask-based approach, wherein 3D-printed facial masks are used, was recently explored by Erdogmus and Marcel [11].

As is the typical case in the field of computer security, the cleverness of skilled, motivated adversaries drove system designers to incorporate defensive techniques in the biometric solutions they develop. This cat-and-mouse game continues to play out in the realm of face authentication systems, and the current recommendation calls for the use of well-designed face liveness detection schemes (that attempt to distinguish a real user from a spoofed one). Indeed, most modern systems now require more active participation compared to simple blink detection, often asking the user to rotate her head or raise an eyebrow during login. Motion-based techniques that check, for example, that the input captured during login exhibits sufficient 3D behavior, are also an active area of research in face authentication.

One such example is the recent work of Li et al. [34] that appeared in CCS’2015. In that work, the use of liveness detection was proposed as a solution to thwarting video-based attacks by checking the consistency of the recorded data with inertial sensors. Such a detection scheme relies on the fact that as a camera moves relative to a user’s stationary head, the facial features it detects will also move in a predictable way. Thus, a 2D video of the victim would have to be captured under the exact same camera motion in order to fool the system.

As mentioned in [34], 3D-printed facial reconstructions offer one option for defeating motion-based liveness detection schemes. In our view, a more realizable approach is to present the system with a 3D facial mesh in a virtual reality (VR) environment. Here, the motion

of the authenticating camera is tracked, and the VR system internally rotates and translates the mesh to match. In this fashion, the camera observes exactly the same movement of facial features as it would for a real face, fulfilling the requirements for liveness detection. Such an attack defeats color-image- and motion-based face authentication on a fundamental level because, with sufficient effort, a VR system can display an environment that is essentially indistinguishable from real-world input.

In this paper, we show that it is possible to undermine modern face authentication systems using one such attack. Moreover, we show that an accurate facial model can be built using *only* a handful of publicly accessible photos — collected, for example, from social network websites — of the victim. From a pragmatic point of view, we are confronted with two main challenges: *i*) the number of photos of the target may be limited, and *ii*) for each available photo, the illumination setting is unknown and the user’s pose and expression are not constrained. To overcome these challenges, we leverage robust, publicly available 3D face reconstruction methods from the field of computer vision, and adapt these techniques to fit our needs. Once a credible synthetic model of a user is obtained, we then employ entry-level virtual reality displays to defeat the state of the art in liveness detection.

The rest of the paper is laid out as follows: §2 provides background and related work related to face authentication, exploitation of users’ online photos, and 3D facial reconstruction. §3 outlines the steps we take to perform our VR-based attack. In §4, we evaluate the performance of our method on 5 commercial face authentication systems and, additionally, on a proposed state-of-the-art system for liveness detection. We suggest steps that could be taken to mitigate our attack in §5, and we address the implications of our successful attack strategy in §6.

2 Background and Related Work

Before delving into the details of our approach, we first present pertinent background information needed to understanding the remainder of this paper.

First, we note that given the three prominent classes of spoofing attacks mentioned earlier, it should be clear that while still-image-based attacks are the easiest to perform, they can be easily countered by detecting the 3D structure of the face. Video-based spoofing is more difficult to accomplish because facial videos of the target user may be harder to come by; moreover, such attacks can also be successfully defeated, for example, using the recently suggested techniques of Li et al. [34] (which we discuss in more detail later). 3D-mask-based approaches, on the other hand, are harder to counter. That said, building a 3D mask is arguably more time-consuming and also requires specialized equipment. Nevertheless, because

²<https://www.youtube.com/watch?v=zYxphDK6s3I>

of the threat this attack vector poses, much research has gone into detecting the textures of 3D masks [11].

2.1 Modern Defenses Against Spoofing

Just as new types of spoofing attacks have been introduced to fool face authentication systems, so too have more advanced methods for countering these attacks been developed. Nowadays, the most popular liveness detection techniques can be categorized as either texture-based approaches, motion-based approaches, or liveness assessment approaches. We discuss each in turn.

Texture-based approaches [11, 25, 37, 40, 54, 60] attempt to identify spoofing attacks based on the assumption that a spoofed face will have a distinctly different texture from a real face. Specifically, they assume that due to properties of its generation, a spoofed face (irrespective of whether it is printed on paper, shown on a display, or made as a 3D mask) will be different from a real face in terms of shape, detail, micro-textures, resolution, blurring, gamma correction, and shading. That is, these techniques rely on perceived limitations of image displays and printing techniques. However, with the advent of high-resolution displays (*e.g.*, 5K), the difference in visual quality between a spoofed image and a living face is hard to notice. Another limitation is that these techniques often require training on every possible spoofing material, which is not practical for real systems.

Motion-based approaches [3, 27, 29, 32, 57] detect spoofing attacks by using motion of the user’s head to infer 3D shape. Techniques such as optical flow and focal-length analysis are typically used. The basic assumption is that structures recovered from genuine faces usually contain sufficient 3D information, whereas structures from fake faces (photos) are usually planar in depth. For instance, the approach of Li et al. [34] checks the consistency of movement between the mobile device’s internal motion sensors and the observed change in head pose computed from the recorded video taken while the claimant attempts to authenticate herself to the device. Such 3D reasoning provides a formidable defense against both still-image and video-based attacks.

Lastly, liveness assessment techniques [19, 30, 31, 49] require the user to perform certain tasks during the authentication stage. For the systems we evaluated, the user is typically asked to follow certain guidelines during registration, and to perform a random series of actions (*e.g.*, eye movement, lip movement, and blinking) at login. The requested gestures help to defeat contemporary spoofing attacks.

Take-away: For real-world systems, liveness detection schemes are often combined with motion-based approaches to provide better security protection than either

can provide on their own. With these ensemble techniques, traditional spoofing attacks can be reliably detected. For that reason, the combination of motion-based systems and liveness detectors has gained traction and is now widely adopted in many commercial systems, including popular face authentication systems offered by companies like KeyLemon, Rohos, and Biomids. For the remainder of this paper, we consider this combination as the state of the art in defenses against spoofing attacks for face authentication systems.

2.2 Online Photos and Face Authentication

It should come as no surprise that personal photos from online social networks can compromise privacy. Major social network sites advise users to set privacy settings for the images they upload, but the vast majority of these photos are often accessible to the public or set to ‘friend-only’ viewing’ [14, 26, 35]. Users also do not have direct control over the accessibility of photos of themselves posted by other users, although they can remove (‘untag’) the association of such photos with their account.

A notable use of social network photos for online security is Facebook’s social authentication (SA) system [15], an extension of CAPTCHAs that seeks to bolster identity verification by requiring the user to identify photos of their friends. While this method does require more specific knowledge than general CAPTCHAs, Polakis et al. [42] demonstrated that facial recognition could be applied to a user’s public photos to discover their social relationships and solve 22% of SA tests automatically.

Given that one’s online photo presence is not entirely controlled by the user alone — but by their collective social circles — many avenues exist for an attacker to uncover the facial appearance of a user, even when the user makes private their own personal photos. In an effort to curb such easy access, work by Ilia et al. [17] has explored the automatic privatization of user data across a social network. This method uses face detection and photo tags to selectively blur the face of a user when the viewing party does not have permission to see the photo. In the future, such an approach may help decrease the public accessibility of users’ personal photos, but it is unlikely that an individual’s appearance can ever be completely obfuscated from attackers across all social media sites and image stores on the Internet.

Clearly, the availability of online user photos is a boon for an adversary tasked with the challenge of undermining face authentication systems. The most germane on this front is the work of Li et al. [33]. There, the authors proposed an attack that defeated commonly used face authentication systems by using photos of the target user gathered from online social networks. Li et al. [33] reported that 77% of the users in their test set were vul-

nerable to their proposed attack. However, their work is targeted at face recognition systems that *do not incorporate face liveness detection*. As noted in §2, in modern face authentication software, sophisticated liveness detection approaches are already in use, and these techniques thwart still-image spoofing attacks of the kind performed by Li et al. [33].

2.3 3D Facial Reconstruction

Constructing a 3D facial model from a small number of personal photos involves the application of powerful techniques from the field of computer vision. Fortunately, there exists a variety of reconstruction approaches that make this task less daunting than it may seem on first blush, and many techniques have been introduced for facial reconstruction from single images [4, 23, 24, 43], videos [20, 48, 51], and combinations of both [52]. For pedagogical reasons, we briefly review concepts that help the reader better understand our approach.

The most popular facial model reconstruction approaches can be categorized into three classes: shape from shading (SFS), structure from motion (SFM) combined with dense stereoscopic depth estimation, and statistical facial models. The SFS approach [24] uses a model of scene illumination and reflectance to recover face structure. Using this technique, a 3D facial model can be reconstructed from only a single input photo. SFS relies on the assumption that the brightness level and gradient of the face image reveals the 3D structure of the face. However, the constraints of the illumination model used in SFS require a relatively simple illumination setting and, therefore, cannot typically be applied to real-world photo samples, where the configuration of the light sources is unknown and often complicated.

As an alternative, the structure from motion approach [12] makes use of multiple photos to triangulate spatial positions of 3D points. It then leverages stereoscopic techniques across the different viewpoints to recover the complete 3D surface of the face. With this method, the reconstruction of a dense and accurate model often requires many consistent views of the surface from different angles; moreover, non-rigid variations (*e.g.*, facial expressions) in the images can easily cause SFM methods to fail. In our scenario, these requirements make such an approach less usable: for many individuals, only a limited number of images might be publicly available online, and the dynamic nature of the face makes it difficult to find multiple images having a consistent appearance (*i.e.*, the exact same facial expression).

Unlike SFS and SFM, statistical facial models [4, 43] seek to perform facial reconstruction on an image using a training set of existing facial models. The basis for this type of facial reconstruction is the 3D morphable model

(3DMM) of Blanz and Vetter [6, 7], which learns the principal variations of face shape and appearance that occur within a population, then fits these properties to images of a specific face. Training the morphable models can be performed either on a controlled set of images [8, 39] or from internet photo-collections [23]. The underlying variations fall on a continuum and capture both expression (*e.g.*, a frowning-to-smiling spectrum) and identity (*e.g.*, a skinny-to-heavy or a male-to-female spectrum). In 3DMM and its derivatives, both 3D shape and texture information are cast into a high-dimensional linear space, which can be analyzed with principal component analysis (PCA) [22]. By optimizing over the weights of different eigenvectors in PCA, any particular human face model can be approximated. Statistical facial models have shown to be very robust and only require a few photos for high-precision reconstruction. For instance, the approach of Baumberger et al. [4] achieves good reconstruction quality using only two images.

To make the process fully automatic, recent 3D facial reconstruction approaches have relied on a few facial landmark points instead of operating on the whole model. These landmarks can be accurately detected using the supervised descent method (SDM) [59] or deep convolutional networks [50]. By first identifying these 2D features in an image and then mapping them to points in 3D space, the entire 3D facial surface can be efficiently reconstructed with high accuracy. In this process, the main challenge is the localization of facial landmarks within the images, especially contour landmarks (along the cheekbones), which are half-occluded in non-frontal views; we introduce a new method for solving this problem when multiple input images are available.

The end result of 3D reconstruction is a untextured (*i.e.*, lacking skin color, eye color, etc.) facial surface. Texturing is then applied using source image(s), creating a realistic final face model. We next detail our process for building such a facial model from a user’s publicly available internet photos, and we outline how this model can be leveraged for a VR-based face authentication attack.

3 Our Approach

A high-level overview of our approach for creating a synthetic face model is shown in Figure 1. Given one or more photos of the target user, we first automatically extract the landmarks of the user’s face (stage ①). These landmarks capture the pose, shape, and expression of the user. Next, we estimate a 3D facial model for the user, optimizing the geometry to match the observed 2D landmarks (stage ②). Once we have recovered the shape of the user’s face, we use a single image to transfer texture information to the 3D mesh. Transferring the texture is non-trivial since parts of the face might be self-occluded

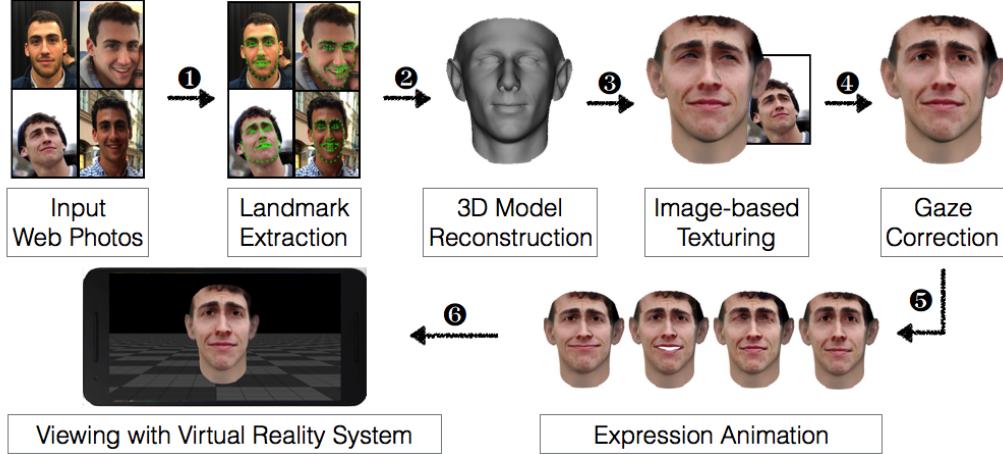


Figure 1: Overview of our proposed approach.

(e.g., when the photo is taken from the side). The texture of these occluded parts must be estimated in a manner that does not introduce too many artifacts (stage ③). Once the texture is filled, we have a realistic 3D model of the user’s face based on a single image.

However, despite its realism, the output of stage ③ is still not able to fool modern face authentication systems. The primary reason for this is that modern face authentication systems use the subject’s gaze direction as a strong feature, requiring the user to look at the camera in order to pass the system. Therefore, we must also automatically correct the direction of the user’s gaze on the textured mesh (stage ④). The adjusted model can then be deformed to produce animation for different facial expressions, such as smiling, blinking, and raising the eyebrows (stage ⑤). These expressions are often used as liveness clues in face authentication systems, and as such, we need to be able to automatically reproduce them on our 3D model. Finally, we output the textured 3D model into a virtual reality system (stage ⑥).

Using this framework, an adversary can bypass both the face recognition and liveness detection components of modern face authentication systems. In what follows, we discuss the approach we take to solve each of the various challenges that arise in our six-staged process.

3.1 Facial Landmark Extraction

Starting from multiple input photos of the user, our first task is to perform facial landmark extraction. Following the approach of Zhu et al. [63], we extract 68 2D facial landmarks in each image using the supervised descent method (SDM) [59]. SDM successfully identifies facial landmarks under relatively large pose differences (± 45 deg yaw, ± 90 deg roll, ± 30 deg pitch). We chose the technique of Zhu et al. [63] because it achieves a me-

dian alignment error of 2.7 pixels on well-known datasets [1] and outperforms other commonly used techniques (e.g., [5]) for landmark extraction.

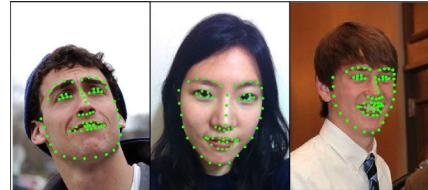


Figure 2: Examples of facial landmark extraction

For our needs, SDM works well on most online images, even those where the face is captured at a low resolution (e.g., 40×50 pixels). It does, however, fail on a handful of the online photos we collected (less than 5%) where the pose is beyond the tolerance level of the algorithm. If this occurs, we simply discard the image. In our experiments, the landmark extraction results are manually checked for correctness, although an automatic scoring system could potentially be devised for this task. Example landmark extractions are shown in Figure 2.

3.2 3D Model Reconstruction

The 68 extracted 3D point landmarks from each of the N input images provide us with a set of coordinates $s_{i,j} \in \mathbb{R}^2$, with $1 \leq i \leq 68, 1 \leq j \leq N$. The projection of the 3D points $S_{i,j} \in \mathbb{R}^3$ on the face onto the image coordinates $s_{i,j}$ follows what is called the “weak perspective projection” (WPP) model [16], computed as follows:

$$s_{i,j} = f_j P R_j (S_{i,j} + t_j), \quad (1)$$

where f_j is a uniform scaling factor; P is the projection matrix $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$; R_j is a 3×3 rotation matrix defined by

the pitch, yaw, and roll, respectively, of the face relative to the camera; and $t_j \in \mathbb{R}^3$ is the translation of the face with respect to the camera. Among these parameters, only $s_{i,j}$ and P are known, and so we must estimate the others.

Fortunately, a large body of work exists on the shape statistics of human faces. Following Zhu et al. [63], we capture face characteristics using the 3D Morphable Model (3DMM) [39] with an expression extension proposed by Chu et al. [9]. This method characterizes variations in face shape for a population using principal component analysis (PCA), with each individual’s 68 3D point landmarks being concatenated into a single feature vector for the analysis. These variations can be split into two categories: constant factors related to an individual’s distinct appearance (identity), and non-constant factors related to expression. The identity axes capture characteristics such as face width, brow placement, or lip size, while the expression axes capture variations like smiling versus frowning. Example axes for variations in expression are shown in Figure 6.

More formally, for any given individual, the 3D coordinates $S_{i,j}$ on the face can be modeled as

$$S_{i,j} = \bar{S}_i + A_i^{id} \alpha_i^{id} + A_i^{exp} \alpha_j^{exp}, \quad (2)$$

where \bar{S}_i is the statistical average of $S_{i,j}$ among the individuals in the population, A_i^{id} is the set of principal axes of variation related to identity, and A_i^{exp} is the set of principal axes related to expression. α_i^{id} and α_j^{exp} are the identity and expression weight vectors, respectively, that determine *person-specific* facial characteristics and expression-specific facial appearance. We obtain \bar{S}_i and A_i^{id} using the 3D Morphable Model [39] and A_i^{exp} from Face Warehouse [8].

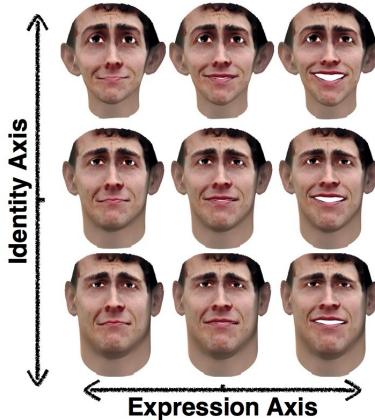


Figure 3: Illustration of identity axes (heavy-set to thin) and expression axes (pursed lips to open smile).

When combining Eqs. (1) and (2), we inevitably run into the so-called “correspondence problem.” That is,

given each identified facial landmark $s_{i,j}$ in the input image, we need to find the corresponding 3D point $S_{i',j}$ on the underlying face model. For landmarks such as the corners of the eyes and mouth, this correspondence is self-evident and consistent across images. However, for contour landmarks marking the edge of the face in an image, the associated 3D point on the user’s facial model is pose-dependent: When the user is directly facing the camera, their jawline and cheekbones are fully in view, and the observed 2D landmarks lie on the fiducial boundary on the user’s 3D facial model. When the user rotates their face left (or right), however, the previously observed 2D contour landmarks on the left (resp. right) side of the face shift out of view. As a result, the observed 2D landmarks on the edge of the face correspond to 3D points closer to the center of the face. This 3D point displacement must be taken into account when recovering the underlying facial model.

Qu et al. [44] deal with contour landmarks using constraints on surface normal direction, based on the observation that points on the edge of the face in the image will have surface normals perpendicular to the viewing direction. However, this approach is less robust because the normal direction cannot always be accurately estimated and, as such, requires careful parameter tuning. Zhu et al. [63] proposed a “landmark marching” scheme that iteratively estimates 3D head pose and 2D contour landmark position. While their approach is efficient and robust against different face angles and surface shapes, it can only handle a single image and cannot refine the reconstruction result using additional images.

Our solution to the correspondence problem is to model 3D point variance for each facial landmark using a pre-trained Gaussian distribution (see Appendix A). Unlike the approach of Zhu et al. [63] which is based on single image input, we solve for pose, perspective, expression, and neutral-expression parameters over *all* images jointly. From this, we obtain a neutral-expression model S_i of the user’s face. A typical reconstruction, S_i , is presented in Figure 4.

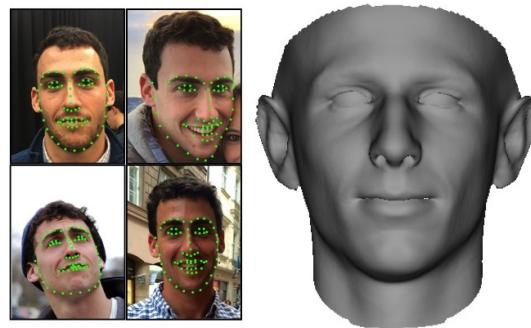


Figure 4: 3D facial model (right) built from facial landmarks extracted from 4 images (left).

3.3 Facial Texture Patching

Given the 3D facial model, the next step is to patch the model with realistic textures that can be recognized by the face authentication systems. Due to the appearance variation across social media photos, we have to achieve this by mapping the pixels in a *single* captured photo onto the 3D facial model, which avoids the challenges of mixing different illuminations of the face. However, this still leaves many of the regions without texture, and those untextured spots will be noticeable to modern face authentication systems. To fill these missing regions, the naïve approach is to utilize the vertical symmetry of the face and fill the missing texture regions with their symmetrical complements. However, doing so would lead to strong artifacts at the boundary of missing regions. A realistic textured model should be free of these artifacts.

To lessen the presence of these artifacts, one approach is to iteratively average the color of neighboring vertices as a color trend and then mix this trend with texture details [45]. However, such an approach over-simplifies the problem and fails to realistically model the illumination of facial surfaces. Instead, we follow the suggestion of Zhu et al. [63] and estimate facial illumination using spherical harmonics [61], then fill in texture details with Poisson editing [41]. In this way, the output model will appear to have a more natural illumination. Sadly, we cannot use their approach directly as it reconstructs a planar normalized face, instead of a 3D facial model, and so we must extend their technique to the 3D surface mesh.

The idea we implemented for improving our initial textured 3D model was as follows: Starting from the single photo chosen as the main texture source, we first estimate and subsequently remove the illumination conditions present in the photo. Next, we map the textured facial model onto a plane via a conformal mapping, then impute the unknown texture using 2D Poisson editing. We further extend their approach to three dimensions and perform Poisson editing directly on the surface of the facial model. Intuitively, the idea behind Poisson editing is to keep the detailed texture in the editing region while enforcing the texture’s smoothness across the boundary. This process is defined mathematically as

$$\Delta f = \Delta g, s.t. f|_{\partial\Omega} = f^0|_{\partial\Omega}, \quad (3)$$

where Ω is the editing region, f is the editing result, f^0 is the known original texture value, and g is the texture value in the editing region that is unknown and needs to be patched with its reflection complement. On a 3D surface mesh, every vertex is connected with 2 to 8 neighbors. Transforming Eq. 3 into discrete form, we have

$$|N_p|f_p - \sum_{q \in N_p \cap \Omega} f_q = \sum_{q \in N_p \cap \bar{\Omega}} f_q^0 + (\Delta g)_p, \quad (4)$$

where N_p is the neighborhood of point p on the mesh. Our enhancement is a natural extension of the Poisson editing method suggested in the seminal work of Pérez et al. [41], although no formulation was given for 3D. By solving Eq. 4 instead of projecting the texture onto a plane and solving Eq. 3, we obtain more realistic texture on the facial model, as shown in Figure 5.



Figure 5: Naïve symmetrical patching (left); Planar Poisson editing (middle); 3D Poisson editing (right).

3.4 Gaze Correction

We now have a realistic 3D facial model of the user. Yet, we found that models at stage ③ were unable to bypass most well-known face recognition systems. Digging deeper into the reasons why, we observed that most recognition systems rely heavily on gaze direction during authentication, *i.e.*, they fail-close if the user is not looking at the device. To address this, we introduce a simple, but effective, approach to correct the gaze direction of our synthetic model (Figure 1, Stage ④).

The idea is as follows. Since we have already reconstructed the texture of the facial model, we can synthesize the texture data in the eye region. These data contain the color information from the sclera, cornea, and pupil and form a three-dimensional distribution in the RGB color space. We estimate this color distribution with a 3D Gaussian function whose three principle components can be computed as (b_1, b_2, b_3) with weight $(\sigma_1, \sigma_2, \sigma_3)$, $\sigma_1 \geq \sigma_2 \geq \sigma_3 > 0$. We perform the same analysis for the eye region of the average face model obtained from 3DMM [39], whose eye is looking straight towards the camera, and we similarly obtain principle color components $(b_1^{std}, b_2^{std}, b_3^{std})$ with weight $(\sigma_1^{std}, \sigma_2^{std}, \sigma_3^{std})$, $\sigma_1^{std} \geq \sigma_2^{std} \geq \sigma_3^{std} > 0$. Then, we convert the eye texture from the average model into the eye texture of the user. For a texture pixel c in the eye region of average texture, we convert it to

$$c_{convert} = \sum_{i=1}^3 \frac{\sigma_i}{\sigma_i^{std}} (c' b_i^{std}) b_i. \quad (5)$$

In effect, we align the color distribution of the average eye texture with the color distribution of the user’s eye texture. By patching the eye region of the facial model with this converted average texture, we realistically capture the user’s eye appearance with forward gaze.

3.5 Adding Facial Animations

Some of the liveness detection methods that we test require that the user performs specific actions in order to unlock the system. To mimic these actions, we can simply animate our facial model using a pre-defined set of facial expressions (*e.g.*, from FaceWarehouse [8]). Recall that in deriving in Eq. 2, we have already computed the weight for the identity axis α_{std}^{id} , which captures the user-specific face structure in a neutral expression. We can adjust the expression of the model by substituting a specific, known expression weight vector α_{std}^{exp} into Eq. 2. By interpolating the model’s expression weight from 0 to α_{std}^{exp} , we are able to animate the 3D facial model to smile, laugh, blink, and raise the eyebrows (see Figure 6).



Figure 6: Animated expressions. From left to right: smiling, laughing, closing the eyes, and raising the eyebrows.

3.6 Leveraging Virtual Reality

While the previous steps were necessary to recover a realistic, animated model of a targeted user’s face, our driving insight is that virtual reality systems can be leveraged to display this model as if it were a real, three-dimensional face. This VR-based spoofing constitutes a fundamentally new class of attacks that exploit weaknesses in camera-based authentication systems.

In the VR system, the synthetic 3D face of the user is displayed on the screen of the VR device, and as the device rotates and translates in the real world, the 3D face moves accordingly. To an observing face authentication system, the depth and motion cues of the display exactly match what would be expected for a human face. Our experimental VR setup consists of custom 3D-rendering software displayed on a Nexus 5X smart phone. Given the ubiquity of smart phones in modern society, our implementation is practical and comes at no additional hardware cost to an attacker. In practice, any device with similar rendering capabilities and inertial sensors could be used.

On smart phones, accelerometers and gyroscopes work in tandem to provide the device with a sense of self-motion. An example use case is detecting when the device is rotated from a portrait view to a landscape view, and rotating the display, in response. However, these sensors are not able to recover absolute *translation* — that is, the device is unable to determine how its position has

changed in 3D space. This presents a challenge because without knowledge of how the device has moved in 3D space, we cannot move our 3D facial model in a realistic fashion. As a result, the observed 3D facial motion will not agree with the device’s inertial sensors, causing our method to fail on methods like that of Li et al. [34] that use such data for liveness detection.

Fortunately, it is possible to track the 3D position of a moving smart phone using its outward-facing camera with structure from motion (see §2.3). Using the camera’s video stream as input, the method works by tracking points in the surrounding environment (*e.g.*, the corners of tables) and then estimating their position in 3D space. At the same time, the 3D position of the camera is recovered relative to the tracked points, thus inferring the camera’s change in 3D position. Several computer vision approaches have been recently introduced to solve this problem accurately and in real time on mobile devices [28, 46, 55, 56]. In our experiments, we make use of a printed marker³ placed on a wall in front of the camera, rather than tracking arbitrary objects in the surrounding scene; however, the end result is the same. By incorporating this module into our proof of concept, the perspective of the viewed model due to camera translation can be simulated with high consistency and low latency.⁴

An example setup for our attack is shown in Figure 7. The VR system consists of a Nexus 5X unit using its outward-facing camera to track a printed marker in the environment. On the Nexus 5X screen, the system displays a 3D facial model whose perspective is always consistent with the spatial position and orientation of the authenticating device. The authenticating camera views the facial model on the VR display, and it is successfully duped into believing it is viewing the real face of the user.

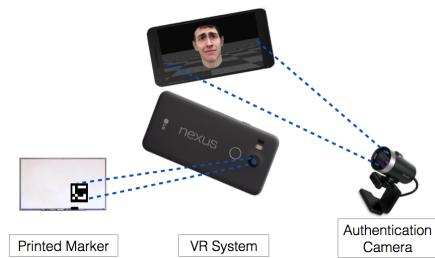


Figure 7: Example setup using virtual reality to mimic 3D structure from motion. The authentication system observes a virtual display of a user’s 3D facial model that rotates and translates as the device moves. To recover the 3D translation of the VR device, an outward-facing camera is used to track a marker in the surrounding environment.

³See Goggle Paper at <http://gogllepaper.com>

⁴Specialized VR systems such as the Oculus Rift could be used to further improve the precision and latency of camera tracking. Such advanced, yet easily obtainable, hardware has the potential to deliver even more sophisticated VR attacks compared to what is presented here.

4 Evaluation

We now demonstrate that our proposed spoofing method constitutes a significant security threat to modern face authentication systems. Using real social media photos from consenting users, we successfully broke five commercial authentication systems with a practical, end-to-end implementation of our approach. To better understand the threat, we further systematically run lab experiments to test the capabilities and limitations of our proposed method. Moreover, we successfully test our proposed approach with the latest motion-based liveness detection approach by Li et al. [34], which is not yet available in commercial systems.

Participants

We recruited 20 volunteers for our tests of commercial face authentication systems. The volunteers were recruited by word of mouth and span graduate students and faculty in two separate research labs. Consultation with our IRB departmental liaison revealed that no application was needed. There was no compensation for participating in the lab study. The ages of the participants range between 24 and 44 years, and the sample consists of 6 females and 14 males. The participants come from a variety of ethnic backgrounds (as stated by the volunteers): 6 are of Asian descent, 4 are Indian, 1 is African-American, 1 is Hispanic, and 8 are Caucasian. With their consent, we collected public photos from the users' Facebook and Google+ social media pages; we also collected any photos we could find of the users on personal or community web pages, as well as via image search on the web. The smallest number of photos we collected for an individual was 3, and the largest number was 27. The average number of photos was 15, with a standard deviation of approximately 6 photos. No private information about the subjects was recorded beside storage of the photographs they consented too. Any images of subjects displayed in this paper was done with the consent of that particular volunteer.

For our experiments, we manually extracted the region around user's face in each image. An adversary could also perform this action automatically using tag information on social media sites, when available. One interesting aspect of social media photos is they may capture significant physical changes of users over time. For instance, one of our participants lost 20 pounds in the last 6 months, and our reconstruction had to utilize images from before and after this change. Two other users had frequent changes in facial hair styles – beards, moustaches, and clean-shaven – all of which we used for our reconstruction. Another user had only uploaded 2 photos to social media in the past 3 years. These varieties all

present challenges for our framework, both for initially reconstructing the user's face and for creating a likeness that matches their current appearance.

Industry-leading Solutions

We tested our approach on five advanced commercial face authentication systems: KeyLemon⁵, Mobius⁶, True Key [18], BioID [21], and 1U App⁷. Table 1 summarizes the training data required by each system when learning a user's facial appearance, as well as the approximate number of users for each system, when available. All systems incorporate some degree of liveness detection into their authentication protocol. KeyLemon and the 1U App require users to perform an action such as blinking, smiling, rotating the head, and raising the eyebrows. In addition, the 1U App requests these actions in a random fashion, making it more resilient to video-based attacks. BioID, Mobius and True Key are motion-based systems and detect 3D facial structure as the user turns their head. It is also possible that these five systems employ other advanced liveness detection approaches, such as texture-based detection schemes, but such information has not been made available to the public.

Methodology

System	Training Method	# Installs
KeyLemon ³	Single video	~100,000
Mobius ²	10 still images	18 reviews
True Key ¹	Single video	50,000-100,000
BioID ²	4 videos	unknown
1U App ¹	1 still image	50-100

Table 1: Summary of the face authentication systems evaluated. The second column lists how each system acquires training data for learning a user's face, and the third column shows the number approximate number of installations or reviews each system has received according to (1) the Google Play Store, (2) the iTunes store, or (3) softpedia.com. BioID is a relatively new app and does not yet have customer reviews on iTunes.

All participants were registered with the 5 face authentication systems under indoor illumination. The average length of time spent by each of the volunteers to register across all systems was 20 minutes. As a control, we first verified that all systems were able to correctly identify the users in the same environment. Next, before testing our method using textures obtained via social media, we evaluated whether our system could spoof the recognition systems using photos taken in this environment. We

⁵<http://www.keylemon.com>

⁶<http://www.biomids.com>

⁷<http://www.1uapps.com>

thus captured one front-view photo for each user under the same indoor illumination and then created their 3D facial model with our proposed approach. We found that these 3D facial models were able to spoof each of the 5 candidate systems with a 100% success rate, which is shown in the second column of Table 2

Following this, we reconstructed each user’s 3D facial model using the images collected from public online sources. As a reminder, any source image can be used as the main image when texturing the model. Since not all textures will successfully spoof the recognition systems, we created textured reconstructions from all source images and iteratively presented them to the system (in order of what we believed to be the best reconstruction, followed by the second best, and so on) until either authentication succeeded or all reconstructions had been tested.

Findings

We summarize the spoofing success rate for each system in Table 2. Except for the 1U system, all facial recognition systems were successfully spoofed for the majority of participants when using social media photos, and all systems were spoofed using indoor, frontal view photos. Out of our 20 participants, there were only 2 individuals for whom none of the systems was spoofed via the social-media-based attack.

Looking into the social media photos we collected of our participants, we observe a few trends among our results. First, we note that moderate- to high-resolution photos lend substantial realism to the textured models. In particular, photos taken by professional photographers (*e.g.*, wedding photos or family portraits) lead to high-quality facial texturing. Such photos are prime targets for facial reconstruction because they are often posted by other users and made publicly available. Second, we note that group photos provide consistent frontal views of individuals, albeit with lower resolution. In cases where high-resolution photos are not available, such frontal views can be used to accurately recover a user’s 3D facial structure. These photos are easily accessible via friends of users, as well. Third, we note that the least spoofable users were not those who necessarily had a low number of personal photos, but rather users who had few forward-facing photos and/or no photos with sufficiently high resolution. From this observation, it seems that creating a realistic texture for user recognition is the primary factor in determining whether a face authentication method will be fooled by our approach. Only a small number of photos are necessary in order to defeat facial recognition systems.

We found that our failure to spoof the 1U App, as well as our lower performance on BioID, using social media photos was directly related to the poor usability of

	Indoor	Social Media	
	Spoof %	Spoof %	Avg. # Tries
KeyLemon	100%	85%	1.6
Mobius	100%	80%	1.5
True Key	100%	70%	1.3
BioID	100%	55%	1.7
1U App	100%	0%	—

Table 2: Success rate for 5 face authentication systems using a model built from (second column) an image of the user taken in an indoor environment and (third and fourth columns) images obtained on users’ social media accounts. The fourth column shows the average number of attempts needed before successfully spoofing the target user.

those systems. Specifically, we found the systems have a very high false rejection rate when live users attempt to authenticate themselves in different illumination conditions. To test this, we had 5 participants register their faces indoors on the 4 mobile systems.⁸ We then had each user attempt to log in to each system 10 times indoors and 10 times outdoors on a sunny day, and we counted the number of accepted logins in each environment for each system. True Key and Mobius, which we found were easier to defeat, correctly authenticated the users 98% and 100% of the time for indoor logins, respectively, and 96% and 100% of the time for outdoor logins. Meanwhile, the indoor/outdoor login rates of BioID and the 1U App were 50%/14% and 96%/48%, respectively. The high false rejection rates under outdoor illumination show that the two systems have substantial difficulty with their authentication when the user’s environment changes. Our impression is that 1U’s single-image user registration simply lacks the training data necessary to accommodate to different illumination settings. BioID is very sensitive to a variety of factors including head rotation and illumination, which leads to many false rejections. (Possibly realizing this, the makers of BioID therefore grant the user 3 trials per login attempt.) Even so, as evidenced by the second column in Table 2, our method still handily defeats the liveness detection modules of these systems given images of the user in the original illumination conditions, which suggests that all the systems we tested are vulnerable to our VR-based attack.

Our findings also suggest that our approach is able to successfully handle significant changes in facial expression, illumination, and for the most part, physical characteristics such as weight and facial hair. Moreover, the approach appears to generalize to users regardless of gender or ethnicity. Given that it has shown to work on a varied collection of real-world data, we believe that the at-

⁸As it is a desktop application, KeyLemon was excluded.

tack presented herein represents a realistic security threat model that could be exploited in the present day.

Next, to gain a deeper understanding of the realism of this threat, we take a closer look at what conditions are necessary for our method to bypass the various face authentication systems we tested. We also consider what main factors contribute to the failure cases of our method.

4.1 Evaluating System Robustness

To further understand the limitations of the proposed spoofing system, we test its robustness against resolution and viewing angle, which are two important factors for the social media photos users upload. Specifically, we answer the question: what is the minimum resolution and maximum head rotation allowed in an uploaded photo before it becomes unusable for spoofing attacks like ours? We further explore how low-resolution frontal images can be used to improve our success rates when high-resolution side-view images are not available.

4.1.1 Blurry, Grainy Pictures Still Say A Lot

To assess our ability to spoof face authentication systems when provided only low-resolution images of a user’s face, we texture the 3D facial models of our sample users using an indoor, frontal view photo. This photo is then downsampled at various resolutions such that the distance between the user’s chin and forehead ranges between 20 and 50 pixels. Then, we attempt to spoof the True Key, BioId, and KeyLemon systems with facial models textured using the down-sampled photos.⁹ If we are successful at a certain resolution, that implies that that resolution leaks the user’s identity information to our spoofing system. The spoofing success rate for various image resolutions is shown in Figure 8.

The result indicates that our approach robustly spoofs face authentication systems when the height of the face in the image is at least 50 pixels. If the resolution of an uploaded photo is less than 30 pixels, the photo is likely of too low-resolution to reliably encode useful features for identifying the user. In our sample set, 88% of users had more than 6 online photos with a chin-to-forehead distance greater than 100 pixels, which easily satisfies the resolution requirement of our proposed spoofing system.

4.1.2 A Little to the Left, a Little to the Right

To identify the robustness of the proposed system against head rotation, we first evaluate the maximum yaw angle allowed for our system to spoof baseline systems using a

⁹We skip analysis of Mobius because its detection method is similar to True Key, and our method did not perform as well on True Key. We also do not investigate the robustness of our method in the 1U system because of our inability to spoof this system using online photos.

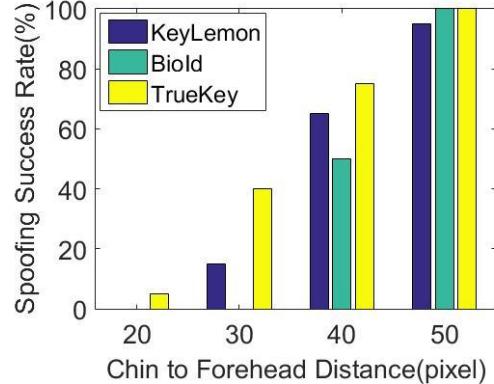


Figure 8: Spoofing success rate with texture taken from photos of different resolution.

single image. For all 20 sample users, we collect multiple indoor photos with yaw angle varying from 5 degrees (approximately frontal view) to 40 degrees (significantly rotated view). We then perform 3D reconstruction for each image, for each user, on the same three face authentication systems. The spoofing success rate for a single input image as a function of head rotation is illustrated in Figure 9 (left). It can be seen that the proposed method successfully spoofs all the baseline systems when the input image has a largely frontal view. As yaw angle increases, it becomes more difficult to infer the user’s frontal view from the image, leading to a decreased spoofing success rate.

4.1.3 For Want of a Selfie

The results of Figure 9 (left) indicate that our success rate falls dramatically if given only a single image with a yaw angle larger than 20 degrees. However, we argue that these high-resolution side-angle views can serve as base images for facial texturing if additional low-resolution frontal views of the user are available. We test this hypothesis by taking, for each user, the rotated images from the previous section along with 1 or 2 low-resolution frontal view photos (chin-to-forehead distance of 30 pixels). We then reconstruct each user’s facial model and use it to spoof our baseline systems. Alone, the provided low-resolution images provide insufficient texture for spoofing, and the higher-resolution side view does not provide adequate facial structure. As shown in Figure 9 (right), by *using the low-resolution front views to guide 3D reconstruction and then using the side view for texturing*, the spoofing success rate for large-angle head rotation increases substantially. From a practical standpoint, low-resolution frontal views are relatively easy to obtain, since they can often be found in publicly posted group photos.

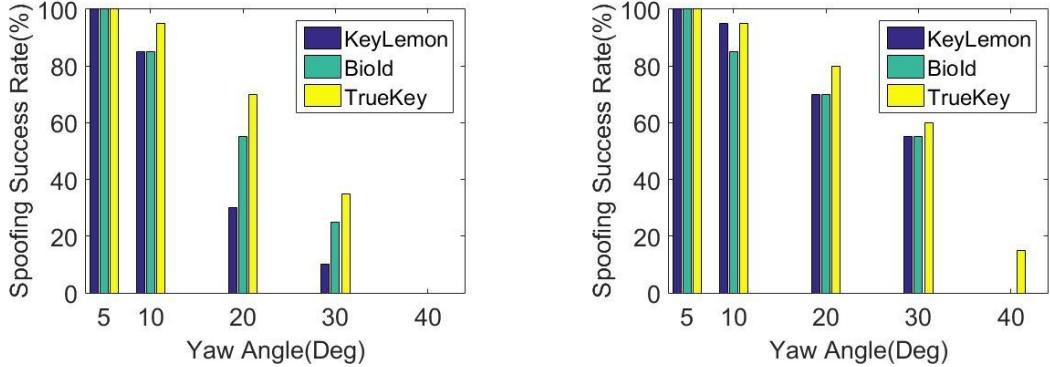


Figure 9: Spoofing success rate with different yaw angles. Left: Using only a single image at the specified angle. Right: Supplementing the single image with low-resolution frontal views, which aid in 3D reconstruction.

4.2 Seeing Your Face Is Enough

Our approach not only defeats existing commercial systems having liveness detection — it fundamentally undermines the process of liveness detection based on color images, entirely. To illustrate this, we use our method to attack the recently proposed authentication approach of Li et al. [34], which obtains a high rate of success in guarding against video-based spoofing attacks. This system adds another layer to motion-based liveness detection by requiring that the movement of the face in the captured video be consistent with the data obtained from the motion sensor of the device. Fortunately, as discussed in §3, the data consistency requirement is automatically satisfied with our virtual reality spoofing system because the 3D model rotates in tandem with the camera motion.

Central to Li et al. [34]’s approach is to build a classifier that evaluates the consistency of captured video and motion sensor data. In turn, the learned classifier is used to distinguish real faces from spoofed ones. Since their code and training samples have not been made public, we implemented our own version of Li et al. [34]’s liveness detection system and trained a classifier with our own training data. We refer the reader to [34] for a full overview of the method.

Following the methodology of [34], we capture video samples (and inertial sensor data) of ~ 4 seconds from the front-facing camera of a mobile phone. In each sample, the phone is held at a distance of 40cm from the subject and moved back-and-forth 20cm to the left and right. We capture 40 samples of real subjects moving the phone in front of their face, 40 samples where a pre-recorded video of a user is presented to the camera, and 30 samples where the camera is presented with a 3D reconstruction of a user in our VR environment. For training, we use a binary logistic regression classifier trained on 20 samples from each class, with the other samples used for testing. Due to the relatively small size of our

training sets, we repeat our classification experiments 4 times, with random train/test splits in each trial, and we report the average performance over all four trials.

Training Data	Real	Video	VR
Real+Video	19.50 / 20	0.25 / 20	9.75 / 10
Real+Video+VR	14.00 / 20	0.00 / 20	5.00 / 10
Real+VR	14.75 / 20	—	5.00 / 10

Table 3: Number of testing samples classified as real users. Values in the first column represent true positive rates, and the second and third columns represent false positives. Each row shows the classification results after training on the classes in the first column. The results were averaged over four trials.

The results of our experiments are shown in Table 3. For each class (real user data, video spoof data, and VR data), we report the average number (over 4 trials) of test samples classified as real user data. We experiment with three different training configurations, which are listed in the first column of the table. The first row shows the results when using real user data as positive samples and video spoof data as negative samples. In this case, it can easily be seen that the real-versus-video identification is almost perfect, matching the results of [34]. However, our VR-based attack is able to spoof this training configuration nearly 100% of the time. The second and third rows of Table 3 show the classification performance when VR spoof data is included in the training data. In both cases, our approach defeats the liveness detector in 50% of trials, and the real user data is correctly identified as such less than 75% of the time.

All three training configurations clearly point to the fact that our VR system presents motion features that are close to real user data. Even if the liveness detector of [34] is specifically trained to look for our VR-based attack, 1 out of every 2 attacks will still succeed, with the false rejection rate also increasing. Any system using

this detector will need to require multiple log-in attempts to account for the decreased recall rate; allowing multiple log-in attempts, however, allows our method more opportunities to succeed. Overall, the results indicate that the proposed VR-based attack successfully spoofs Li et al. [34]’s approach, which is to our knowledge the state of the art in motion-based liveness detection.

5 Defense in Depth

While current facial authentication systems succumb to our VR-based attack, several features could be added to these systems to confound our approach. Here, we detail three such features, namely random projection of light patterns, detection of minor skin tone fluctuations related to pulse, and the use of illuminated infrared (IR) sensors. Of these, the first two could still be bypassed with additional adversary effort, while the third presents a significantly different hardware configuration that would require non-trivial alterations to our method.

Light Projection The principle of using light projection for liveness detection is simple: Using an outward-facing light source (*e.g.*, the flashlight commonly included on camera-equipped mobile phones), flash light on the user’s face at random intervals. If the observed change in illumination does not match the random pattern, then face authentication fails. The simplicity of this approach makes it appealing and easily implementable; however, an adversary could modify our proposed approach to detect the random flashes of light and, with low latency, subsequently add rendered light to the VR scene. Random projections of structured light [62], *i.e.*, checkerboard patterns and lines, would increase the difficulty of such an attack, as the 3D-rendering system must be able to quickly and accurately render the projected illumination patterns on a model. However, structured light projection requires specialized hardware that typically is not found on smart phones and similar devices, which decreases the feasibility of this mitigation.

Pulse Detection Recent computer vision research [2, 58] has explored the prospect of video magnification, which transforms micro-scale fluctuations over time into strong visual changes. One such application is the detection of human pulse from a standard video of a human face. The method detects small, periodic color changes related to pulse in the region of the face and then amplifies this effect such that the face appears to undergo strong changes in brightness and hue. This amplification could be used as an additional method for liveness detection by requiring that the observed face have a detectable pulse. Similar ideas have been applied to fingerprint systems that check for blood flow using light emitted from

beneath a prism. Of course, an attacker using our proposed approach could simply add subtle color variation to the 3D model to approximate this effect. Nevertheless, such a method would provide another layer of defense against spoofed facial models.

Infrared Illumination Microsoft released Windows Hello as a more personal way to sign into Windows 10 devices with just a look or a touch. The new interface supports biometric authentication that includes face, iris, or fingerprint authentication. The platform includes Intel’s RealSense IR-based, rather than a color-based, facial authentication method. In principle, their approach works in the same way as contemporary face authentication methods, but instead uses an IR camera to capture a video of the user’s face. The attack presented in this paper would fail to bypass this approach because typical VR displays are not built to project IR light; however, specialized IR display hardware could potentially be used to overcome this limitation.

One limiting factor that may make IR-based techniques less common (especially on mobile devices) is the requirement for additional hardware to support this enhanced form of face authentication. Indeed, as of this writing, only a handful of personal computers support Windows Hello.¹⁰ Nevertheless, the use of infrared illumination offers intriguing possibilities for the future.

Takeaway In our opinion, it is highly unlikely that robust facial authentication systems will be able to operate using solely web/mobile camera input. Given the widespread nature of high-resolution personal online photos, today’s adversaries have a goldmine of information at their disposal for synthetically creating fake face data. Moreover, even if a system is able to robustly detect a certain type of attack – be it using a paper printout, a 3D-printed mask, or our proposed method – generalizing to all possible attacks will increase the possibility of false rejections and therefore limit the overall usability of the system. The strongest facial authentication systems will need to incorporate non-public imagery of the user that cannot be easily printed or reconstructed (*e.g.*, a skin heat map from special IR sensors).

6 Discussion

Our work outlines several important lessons for both the present state and the future state of security, particularly as it relates to face authentication systems. First, our exploitation of social media photos to perform facial reconstruction underscores the notion that online privacy of one’s appearance is tantamount to online privacy of other personal information, such as age and location.

¹⁰See “[PC platforms that support Windows Hello](#)” for more info.

The ability of an adversary to recover an individual’s facial characteristics through online photos is an immediate and very serious threat, albeit one that clearly cannot be completely neutralized in the age of social media. Therefore, it is prudent that face recognition tools become increasingly robust against such threats in order to remain a viable security option in the future.

At a minimum, it is imperative that face authentication systems be able to reject synthetic faces with low-resolution textures, as we show in our evaluations. Of more concern, however, is the increasing threat of virtual reality, as well as computer vision, as an adversarial tool. It appears to us that the designers of face authentication systems have assumed a rather weak adversarial model wherein attackers may have limited technical skills and be limited to inexpensive materials. This practice is risky, at best. Unfortunately, VR itself is quickly becoming commonplace, cheap, and easy-to-use. Moreover, VR visualizations are increasingly *convincing*, making it easier and easier to create realistic 3D environments that can be used to fool visual security systems. As such, it is our belief that authentication mechanisms of the future must aggressively anticipate and adapt to the rapid developments in the virtual and online realms.

Appendix

A Multi-Image Facial Model Estimation

In §3.2, we outline how to associate 2D facial landmarks with corresponding 3D points on an underlying facial model. Contour landmarks pose a substantial difficulty for this 2D-to-3D correspondence problem because the associated set of 3D points for these features is pose-dependent. Zhu et al. [63] compensate for this phenomenon by modeling contour landmarks with parallel curved line segments and iteratively optimizing head orientation and 2D-to-3D correspondence. For a specific head orientation R_j , the corresponding landmark points on the 3D model are found using an explicit function based on rotation angle:

$$\begin{aligned} s_{i,j} &= f_j PR_j(S_{i',j} + t_j) \\ S_{i',j} &= \bar{S}_{i'} + A_{i'}^{id} \alpha^{id} + A_{i'}^{exp} \alpha_j^{exp} \\ i' &= land(i, R_j), \end{aligned} \quad (6)$$

where $land(i, R_j)$ is the pre-calculated mapping function that computes the position of landmarks i on the 3D model when the orientation is R_j . Ideally, the first equation in Eq. (6) should hold for all the landmark points in all the images. However, this is not the case due to the alignment error introduced by landmark extraction. Generally, contour landmarks introduce more error than

corner landmarks, and this approach actually leads to inferior results when multiple input images are used.

Therefore, different from Zhu et al. [63], we compute the 3D facial model with Maximum a Posteriori (MAP) estimation. We assume the alignment error of each 3D landmark independently follows a Gaussian distribution. Then, the most probable parameters $\theta := (\{f_j\}, \{R_j\}, \{t_j\}, \{\alpha_j^{exp}\}, \alpha^{id})$ can be estimated by minimizing the cost function

$$\begin{aligned} \theta = \operatorname{argmax}_{\theta} \{ &\sum_{i=1}^{68} \sum_{j=1}^N \frac{1}{(\sigma_i^s)^2} \|s_{i,j} - f_j PR_j(S_{i',j} + t_j)\|^2 + \\ &\sum_{j=1}^N (\alpha_j^{exp})' \Sigma_{exp}^{-1} \alpha_j^{exp} + (\alpha^{id})' \Sigma_{id}^{-1} \alpha^{id} \}. \end{aligned} \quad (7)$$

Here, $S_{i',j}$ is computed using Eq. (6). Σ_{id} and Σ_{exp} are covariance matrices of α^{id} and α_j^{exp} , which can be obtained from the pre-existing face model. $(\sigma_i^s)^2$ is the variance of alignment error of the i -th landmark and is obtained from a separate training set consisting 20 images with hand-labeled landmarks. Eq. (7) can be computed efficiently, leading to the estimated identity weight α^{id} , with which we can compute the neutral-expression model $S_i (= \bar{S}_{i'} + A_{i'}^{id} \alpha^{id})$.

References

- [1] S. Baker and I. Matthews. Lucas-kanade 20 years on: A unifying framework. *International Journal of Computer Vision (IJCV)*, 56(3):221–255, 2004.
- [2] G. Balakrishnan, F. Durand, and J. Guttag. Detecting pulse from head motions in video. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3430–3437, 2013.
- [3] W. Bao, H. Li, N. Li, and W. Jiang. A liveness detection method for face recognition based on optical flow field. In *Image Analysis and Signal Processing, International Conference on*, pages 233–236, 2009.
- [4] C. Baumberger, M. Reyes, M. Constantinescu, R. Olariu, E. De Aguiar, and T. Oliveira Santos. 3d face reconstruction from video using 3d morphable model and silhouette. In *Graphics, Patterns and Images (SIBGRAPI), Conference on*, pages 1–8, 2014.
- [5] P. N. Belhumeur, D. W. Jacobs, D. J. Kriegman, and N. Kumar. Localizing parts of faces using a consensus of exemplars. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(12):2930–2940, 2013.
- [6] V. Blanz and T. Vetter. A morphable model for the synthesis of 3d faces. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 187–194. ACM Press/Addison-Wesley Publishing Co., 1999.
- [7] V. Blanz and T. Vetter. Face recognition based on fitting a 3d morphable model. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(9):1063–1074, 2003.

- [8] C. Cao, Y. Weng, S. Zhou, Y. Tong, and K. Zhou. Faceware-house: A 3d facial expression database for visual computing. *Visualization and Computer Graphics, IEEE Transactions on*, 20(3):413–425, 2014.
- [9] B. Chu, S. Romdhani, and L. Chen. 3d-aided face recognition robust to expression and pose variations. In *Computer Vision and Pattern Recognition (CVPR), Conference on*, pages 1907–1914, 2014.
- [10] N. Duc and B. Minh. Your face is not your password. In *Black Hat Conference*, volume 1, 2009.
- [11] N. Erdoganmus and S. Marcel. Spoofing face recognition with 3d masks. *Information Forensics and Security, IEEE Transactions on*, 9(7):1084–1097, 2014.
- [12] D. Fidaleo and G. Medioni. Model-assisted 3d face reconstruction from video. In *Analysis and modeling of faces and gestures*, pages 124–138. Springer, 2007.
- [13] Gartner. Gartner backs biometrics for enterprise mobile authentication. *Biometric Technology Today*, Feb. 2014.
- [14] S. Golder. Measuring social networks with digital photograph collections. In *Proceedings of the nineteenth ACM conference on Hypertext and hypermedia*, pages 43–48, 2008.
- [15] M. Hicks. A continued commitment to security, 2011. URL <https://www.facebook.com/notes/facebook/a-continued-commitment-to-security/486790652130/>.
- [16] R. Horaud, F. Dornaika, and B. Lamiray. Object pose: The link between weak perspective, paraperspective, and full perspective. *International Journal of Computer Vision*, 22(2):173–189, 1997.
- [17] P. Ilia, I. Polakis, E. Athanasopoulos, F. Maggi, and S. Ioannidis. Face/off: Preventing privacy leakage from photos in social networks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 781–792, 2015.
- [18] Intel Security. True Key™ by Intel Security: Security white paper 1.0, 2015. URL <https://b.tkassets.com/shared/TrueKey-SecurityWhitePaper-v1.0-EN.pdf>.
- [19] H.-K. Jee, S.-U. Jung, and J.-H. Yoo. Liveness detection for embedded face recognition system. *International Journal of Biological and Medical Sciences*, 1(4):235–238, 2006.
- [20] L. A. Jeni, J. F. Cohn, and T. Kanade. Dense 3d face alignment from 2d videos in real-time. In *Automatic Face and Gesture Recognition (FG), 2015 11th IEEE International Conference and Workshops on*, volume 1, pages 1–8. IEEE, 2015.
- [21] O. Jesorsky, K. J. Kirchberg, and R. W. Frischholz. Robust face detection using the hausdorff distance. In *Audio-and video-based biometric person authentication*, pages 90–95. Springer, 2001.
- [22] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.
- [23] I. Kemelmacher-Shlizerman. Internet based morphable model. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3256–3263, 2013.
- [24] I. Kemelmacher-Shlizerman and R. Basri. 3D face reconstruction from a single image using a single reference face shape. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(2):394–405, 2011.
- [25] G. Kim, S. Eum, J. K. Suhr, D. I. Kim, K. R. Park, and J. Kim. Face liveness detection based on texture and frequency analyses. In *Biometrics (ICB), 5th IAPR International Conference on*, pages 67–72, 2012.
- [26] H.-N. Kim, A. El Saddik, and J.-G. Jung. Leveraging personal photos to inferring friendships in social network services. *Expert Systems with Applications*, 39(8):6955–6966, 2012.
- [27] S. Kim, S. Yu, K. Kim, Y. Ban, and S. Lee. Face liveness detection using variable focusing. In *Biometrics (ICB), 2013 International Conference on*, pages 1–6, 2013.
- [28] K. Kolev, P. Tanskanen, P. Speciale, and M. Pollefeys. Turning mobile phones into 3d scanners. In *Computer Vision and Pattern Recognition (CVPR), IEEE Conference on*, pages 3946–3953, 2014.
- [29] K. Kollreider, H. Fronthaler, and J. Bigun. Evaluating liveness by face images and the structure tensor. In *Automatic Identification Advanced Technologies, Fourth IEEE Workshop on*, pages 75–80. IEEE, 2005.
- [30] K. Kollreider, H. Fronthaler, M. I. Faraj, and J. Bigun. Real-time face detection and motion analysis with application in liveness assessment. *Information Forensics and Security, IEEE Transactions on*, 2(3):548–558, 2007.
- [31] K. Kollreider, H. Fronthaler, and J. Bigun. Verifying liveness by multiple experts in face biometrics. In *Computer Vision and Pattern Recognition Workshops, IEEE Computer Society Conference on*, pages 1–6, 2008.
- [32] A. Lagorio, M. Tistarelli, M. Cadoni, C. Fookes, and S. Sridharan. Liveness detection based on 3d face shape analysis. In *Biometrics and Forensics (IWBF), International Workshop on*, pages 1–4, 2013.
- [33] Y. Li, K. Xu, Q. Yan, Y. Li, and R. H. Deng. Understanding osn-based facial disclosure against face authentication systems. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 413–424. ACM, 2014.
- [34] Y. Li, Y. Li, Q. Yan, H. Kong, and R. H. Deng. Seeing your face is not enough: An inertial sensor-based liveness detection for face authentication. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 1558–1569, 2015.
- [35] Y. Liu, K. P. Gummadi, B. Krishnamurthy, and A. Mislove. Analyzing facebook privacy settings: user expectations vs. reality. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 61–70. ACM, 2011.
- [36] C. Lu and X. Tang. Surpassing human-level face verification performance on LFW with GaussianFace. *arXiv preprint arXiv:1404.3840*, 2014.
- [37] J. Määttä, A. Hadid, and M. Pietikainen. Face spoofing detection from single images using micro-texture analysis. In *Biometrics (IJCB), International Joint Conference on*, pages 1–7, 2011.
- [38] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep face recognition. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2015.
- [39] P. Paysan, R. Knothe, B. Amberg, S. Romdhani, and T. Vetter. A 3d face model for pose and illumination invariant face recognition. In *Proceedings of the 6th IEEE International Conference on Advanced Video and Signal based Surveillance (AVSS) for Security, Safety and Monitoring in Smart Environments*, 2009.

- [40] B. Peixoto, C. Michelassi, and A. Rocha. Face liveness detection under bad illumination conditions. In *Image Processing (ICIP), 18th IEEE International Conference on*, pages 3557–3560, 2011.
- [41] P. Pérez, M. Gangnet, and A. Blake. Poisson image editing. *ACM Transactions on Graphics (TOG)*, 22(3):313–318, 2003.
- [42] I. Polakis, M. Lancini, G. Kontaxis, F. Maggi, S. Ioannidis, A. D. Keromytis, and S. Zanero. All your face are belong to us: Breaking facebook’s social authentication. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 399–408, 2012.
- [43] C. Qu, E. Monari, T. Schuchert, and J. Beyerer. Fast, robust and automatic 3d face model reconstruction from videos. In *Advanced Video and Signal Based Surveillance (AVSS), 11th IEEE International Conference on*, pages 113–118, 2014.
- [44] C. Qu, E. Monari, T. Schuchert, and J. Beyerer. Adaptive contour fitting for pose-invariant 3d face shape reconstruction. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 1–12, 2015.
- [45] C. Qu, E. Monari, T. Schuchert, and J. Beyerer. Realistic texture extraction for 3d face models robust to self-occlusion. In *IS&T/SPIE Electronic Imaging*. International Society for Optics and Photonics, 2015.
- [46] T. Schops, T. Sattler, C. Hane, and M. Pollefeys. 3d modeling on the go: Interactive 3d reconstruction of large-scale scenes on mobile devices. In *3D Vision (3DV), International Conference on*, pages 291–299, 2015.
- [47] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. *arXiv preprint arXiv:1503.03832*, 2015.
- [48] F. Shi, H.-T. Wu, X. Tong, and J. Chai. Automatic acquisition of high-fidelity facial performances using monocular videos. *ACM Transactions on Graphics (TOG)*, 33(6):222, 2014.
- [49] L. Sun, G. Pan, Z. Wu, and S. Lao. Blinking-based live face detection using conditional random fields. In *Advances in Biometrics*, pages 252–260. Springer, 2007.
- [50] Y. Sun, X. Wang, and X. Tang. Deep convolutional network cascade for facial point detection. In *Computer Vision and Pattern Recognition (CVPR), IEEE Conference on*, pages 3476–3483, 2013.
- [51] S. Suwajanakorn, I. Kemelmacher-Shlizerman, and S. M. Seitz. Total moving face reconstruction. In *Computer Vision–ECCV 2014*, pages 796–812. Springer, 2014.
- [52] S. Suwajanakorn, S. M. Seitz, and I. Kemelmacher-Shlizerman. What makes tom hanks look like tom hanks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3952–3960, 2015.
- [53] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR), IEEE Conference on*, pages 1701–1708, 2014.
- [54] X. Tan, Y. Li, J. Liu, and L. Jiang. Face liveness detection from a single image with sparse low rank bilinear discriminative model. In *European Conference on Computer Vision (ECCV)*, pages 504–517. 2010.
- [55] P. Tanskanen, K. Kolev, L. Meier, F. Camposeco, O. Saurer, and M. Pollefeys. Live metric 3d reconstruction on mobile phones. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 65–72, 2013.
- [56] J. Ventura, C. Arth, G. Reitmayr, and D. Schmalstieg. Global localization from monocular slam on a mobile phone. *Visualization and Computer Graphics, IEEE Transactions on*, 20(4):531–539, 2014.
- [57] T. Wang, J. Yang, Z. Lei, S. Liao, and S. Z. Li. Face liveness detection using 3d structure recovered from a single camera. In *Biometrics (ICB), International Conference on*, pages 1–6, 2013.
- [58] H.-Y. Wu, M. Rubinstein, E. Shih, J. Guttag, F. Durand, and W. T. Freeman. Eulerian video magnification for revealing subtle changes in the world. *ACM Transactions on Graphics (TOG)*, 31(4), 2012.
- [59] X. Xiong and F. De la Torre. Supervised descent method and its applications to face alignment. In *Computer Vision and Pattern Recognition (CVPR), IEEE Conference on*, pages 532–539, 2013.
- [60] J. Yang, Z. Lei, S. Liao, and S. Z. Li. Face liveness detection with component dependent descriptor. In *Biometrics (ICB), International Conference on*, pages 1–6, 2013.
- [61] L. Zhang and D. Samaras. Face recognition from a single training image under arbitrary unknown lighting using spherical harmonics. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(3):351–363, 2006.
- [62] L. Zhang, B. Curless, and S. M. Seitz. Rapid shape acquisition using color structured light and multi-pass dynamic programming. In *3D Data Processing Visualization and Transmission, First International Symposium on*, pages 24–36, 2002.
- [63] X. Zhu, Z. Lei, J. Yan, D. Yi, and S. Z. Li. High-fidelity pose and expression normalization for face recognition in the wild. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 787–796, 2015.

Hidden Voice Commands

Nicholas Carlini*

University of California, Berkeley

Tavish Vaidya

Georgetown University

Clay Shields

Georgetown University

Yuankai Zhang

Georgetown University

David Wagner

University of California, Berkeley

Pratyush Mishra

University of California, Berkeley

Micah Sherr

Georgetown University

Wenchao Zhou

Georgetown University

Abstract

Voice interfaces are becoming more ubiquitous and are now the primary input method for many devices. We explore in this paper how they can be attacked with *hidden voice commands* that are unintelligible to human listeners but which are interpreted as commands by devices.

We evaluate these attacks under two different threat models. In the black-box model, an attacker uses the speech recognition system as an opaque oracle. We show that the adversary can produce difficult to understand commands that are effective against existing systems in the black-box model. Under the white-box model, the attacker has full knowledge of the internals of the speech recognition system and uses it to create attack commands that we demonstrate through user testing are not understandable by humans.

We then evaluate several defenses, including notifying the user when a voice command is accepted; a verbal challenge-response protocol; and a machine learning approach that can detect our attacks with 99.8% accuracy.

1 Introduction

Voice interfaces to computer systems are becoming ubiquitous, driven in part by their ease of use and in part by decreases in the size of modern mobile and wearable devices that make physical interaction difficult. Many devices have adopted an always-on model in which they continuously listen for possible voice input. While voice interfaces allow for increased accessibility and potentially easier human-computer interaction, they are at the same time susceptible to attacks: Voice is a broadcast channel open to any attacker that is able to create sound within the vicinity of a device. This introduces an opportunity for attackers to try to issue unauthorized voice commands to these devices.

An attacker may issue voice commands to any device that is within speaker range. However, naïve attacks will be conspicuous: a device owner who overhears such a

command may recognize it as an unwanted command and cancel it, or otherwise take action. This motivates the question we study in this paper: can an attacker create *hidden voice commands*, i.e., commands that will be executed by the device but which won't be understood (or perhaps even noticed) by the human user?

The severity of a hidden voice command depends upon what commands the targeted device will accept. Depending upon the device, attacks could lead to information leakage (e.g., posting the user's location on Twitter), cause denial of service (e.g., activating airplane mode), or serve as a stepping stone for further attacks (e.g., opening a web page hosting drive-by malware). Hidden voice commands may also be broadcast from a loudspeaker at an event or embedded in a trending YouTube video, compounding the reach of a single attack.

Vaidya et al. [41] showed that hidden voice commands are possible—attackers can generate commands that are recognized by mobile devices but are considered as noise by humans. Building on their work, we show more powerful attacks and then introduce and analyze a number of candidate defenses.

The contributions of this paper include the following:

- We show that hidden voice commands can be constructed even with very little knowledge about the speech recognition system. We provide a general attack procedure for generating commands that are likely to work with any modern voice recognition system. We show that our attacks work against Google Now's speech recognition system and that they improve significantly on previous work [41].
- We show that adversaries with significant knowledge of the speech recognition system can construct hidden voice commands that humans cannot understand at all.
- Finally, we propose, analyze, and evaluate a suite of detection and mitigation strategies that limit the effects of the above attacks.

Audio files for the hidden voice commands and a video demonstration of the attack are available at <http://hiddenvoicecommands.com>.

* Authors listed alphabetically, with student authors appearing before faculty authors.

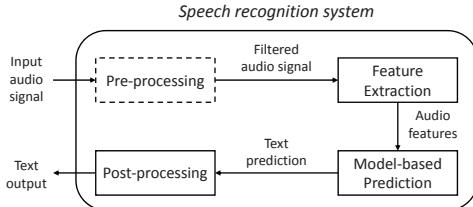


Figure 1: Overview of a typical speech recognition system.

2 Background and Related Work

To set the stage for the attacks that we present in §3 and §4, we briefly review how speech recognition works.

Figure 1 presents a high-level overview of a typical speech recognition procedure, which consists of the following four steps: pre-processing, feature extraction, model-based prediction, and post-processing. Pre-processing performs initial speech/non-speech identification by filtering out frequencies that are beyond the range of a human voice and eliminating time periods where the signal energy falls below a particular threshold. This step only does rudimentary filtering, but still allows non-speech signals to pass through the filter if they pass the energy-level and frequency checks.

The second step, feature extraction, splits the filtered audio signal into short (usually around 20 ms) frames and extracts features from each frame. The feature extraction algorithm used in speech recognition is almost always the Mel-frequency cepstral (MFC) transform [20, 42]. We describe the MFC transform in detail in Appendix A, but at a high level it can be thought of as a transformation that extracts the dominant frequencies from the input.

The model-based prediction step takes as input the extracted features, and matches them against an existing model built offline to generate text predictions. The technique used in this step can vary widely: some systems use Hidden Markov Models, while many recent systems have begun to use recurrent neural networks (RNNs).

Finally, a post-processing step ranks the text predictions by employing additional sources of information, such as grammar rules or locality of words.

Related work. Unauthorized voice commands have been studied by Diao et al. [12] and Jang et al. [21] who demonstrate that malicious apps can inject synthetic audio or play commands to control smartphones. Unlike in this paper, these attacks use non-hidden channels that are understandable by a human listener.

Similar to our work, Kasmi and Lopes Esteves [23] consider the problem of *covert* audio commands. There, the authors inject voice commands by transmitting FM signals that are received by a headset. In our work, we do not require the device to have an FM antenna (which is not often present) and we obfuscate the voice com-

mand so that it is not human-recognizable. Schlegel et al. [36] show that malicious apps can eavesdrop and record phone calls to extract sensitive information. Our work differs in that it exploits targeted devices’ existing functionality (i.e., speech recognition) and does not require the installation of malicious apps.

Earlier work by Vaidya et al. [41] introduces obfuscated voice commands that are accepted by voice interfaces. Our work significantly extends their black-box approach by (i) evaluating the effectiveness of their attacks under realistic scenarios, (ii) introducing more effective “white-box” attacks that leverage knowledge of the speech recognition system to produce machine-understandable speech that is almost never recognized by humans, (iii) formalizing the method of creating hidden voice commands, and (iv) proposing and evaluating defenses.

Image recognition systems have been shown to be vulnerable to attacks where slight modifications to only a few pixels can change the resulting classification dramatically [17, 19, 25, 38]. Our work has two key differences. First, feature extraction for speech recognition is significantly more complex than for images; this is one of the main hurdles for our work. Second, attacks on image recognition have focused on the case where the adversary is allowed to directly modify the electronic image. In contrast, our attacks work “over the air”; that is, we create audio that when played and recorded is recognized as speech. The analogous attack on image recognition systems would be to create a physical object which appears benign, but when photographed, is classified incorrectly. As far as we know, no one has demonstrated such an attack on image recognition systems.

More generally, our attacks can be framed as an evasion attack against machine learning classifiers: if f is a classifier and A is a set of acceptable inputs, given a desired class y , the goal is to find an input $x \in A$ such that $f(x) = y$. In our context, f is the speech recognition system, A is a set of audio inputs that a human would not recognize as speech, and y is the text of the desired command. Attacks on machine learning have been studied extensively in other contexts [1, 4, 5, 10, 13, 22, 31, 40]; In particular, Fawzi et al. [14] develop a rigorous framework to analyze the vulnerability of various types of classifiers to adversarial perturbation of inputs. They demonstrate that a minimal set of adversarial changes to input data is enough to fool most classifiers into misclassifying the input. Our work is different in two key respects: (i) the above caveats for image recognition systems still apply, and moreover, (ii) their work does not necessarily aim to create inputs that are misclassified into a particular category; but rather that it is just misclassified. On the other hand, we aim to craft inputs that are recognized as potentially sensitive commands.

Finally, Fredrikson et al. [15] attempt to invert machine learning models to learn private and potentially sensitive data in the training corpus. They formulate their task as an optimization problem, similar to our white-box approach, but they (i) test their approach primarily on image recognition models, which, as noted above, are easier to fool, and (ii) do not aim to generate adversarial inputs, but rather only extract information about individual data points.

3 Black-box Attacks

We first show that under a weak set of assumptions an attacker with no internal knowledge of a voice recognition system can generate hidden voice commands that are difficult for human listeners to understand. We refer to these as *obfuscated commands*, in contrast to unmodified and understandable *normal commands*.

These attacks were first proposed by Vaidya et al. [41]. This section improves upon the efficacy and practicality of their attacks and analysis by (i) carrying out and testing the performance of the attacks under more practical settings, (ii) considering the effects of background noise, and (iii) running the experiments against Google’s improved speech recognition service [34].

3.1 Threat model & attacker assumptions

In this black-box model the adversary does not know the specific algorithms used by the speech recognition system. We assume that the system extracts acoustic information through some transform function such as an MFC, perhaps after performing some pre-processing such as identifying segments containing human speech or removing noise. MFCs are commonly used in current-generation speech recognition systems [20, 42], making our results widely applicable, but not limited to such systems.

We treat the speech recognition system as an oracle to which the adversary can pose transcription tasks. The adversary can thus learn how a particular obfuscated audio signal is interpreted. We do not assume that a particular transcription is guaranteed to be consistent in the future. This allows us to consider speech recognition systems that apply randomized algorithms as well as to account for transient effects such as background noise and environmental interference.

Conceptually, this model allows the adversary to iteratively develop obfuscated commands that are increasingly difficult for humans to recognize while ensuring, with some probability, that they will be correctly interpreted by a machine. This trial-and-error process occurs in advance of any attack and is invisible to the victim.

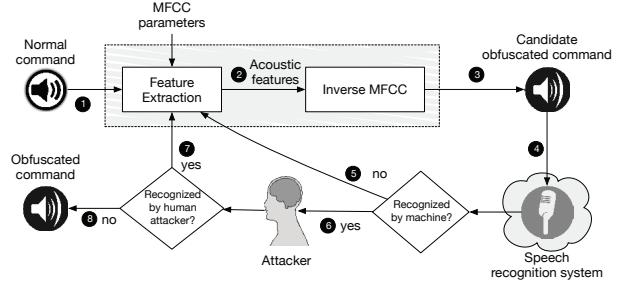


Figure 2: Adversary’s workflow for producing an obfuscated audio command from a normal command.

3.2 Overview of approach

We rerun the black-box attack proposed by Vaidya et al. [41] as shown in Figure 2. The attacker’s goal is to produce an obfuscated command that is accepted by the victim’s speech recognition system but is indecipherable by a human listener.

The attacker first produces a normal command that it wants executed on the targeted device. To thwart individual recognition the attacker may use a text-to-speech engine, which we found is generally correctly transcribed. This command is then provided as input (Figure 2, step ①) to an *audio mangler*, shown as the grey box in the figure. The audio mangler performs an MFC with a starting set of parameters on the input audio, and then performs an inverse MFC (step ②) that additionally adds noise to the output. By performing the MFC and then inverting the obtained acoustic features back into an audio sample, the attacker is in essence attempting to remove all audio features that are not used in the speech recognition system but which a human listener might use for comprehension.

Since the attacker does not know the MFC features used by the speech recognition system, experimentation is required. First, the attacker provides the *candidate obfuscated audio* that results from the MFC→inverse-MFC process (step ③) to the speech recognition system (step ④). If the command is not recognized then the attacker must update the MFC parameters to ensure that the result of the MFC→inverse-MFC transformation will yield higher fidelity audio (step ⑤).

If the candidate obfuscated audio is interpreted correctly (step ⑥), then the human attacker tests if it is human understandable. This step is clearly subjective and, worse, is subject to *priming* effects [28] since the attacker already knows the correct transcription. The attacker may solicit outside opinions by crowdsourcing. If the obfuscated audio is too easily understood by humans the attacker discards the candidate and generates new candidates by adjusting the MFC parameters to produce lower fidelity audio (step ⑦). Otherwise, the can-

Table 1: MFC parameters tuned to produce obfuscated audio.

Parameter	Description
wintime	time for which the signal is considered constant
hoptime	time step between adjacent windows
numcep	number of cepstral coefficients
nbands	no. of warped spectral bands for aggregating energy levels

didate obfuscated audio command—which is recognized by machines but not by humans—is used to conduct the actual attack (step ⑧).

3.3 Experimental setup

We obtained the audio mangling program used by Vaidya et al. [41]. Conforming to their approach, we also manually tune four MFC parameters to mangle and test audio using the workflow described in §3.2 to determine the ranges for human and machine perception of voice commands. The list of modified MFC parameters is presented in Table 1.

Our voice commands consisted of the phrases “OK google”, “call 911”, and “turn on airplane mode”. These commands were chosen to represent a variety of potential attacks against personal digital assistants. Voice commands were played using Harmon Kardon speakers, model number HK695-01,13, in a conference room measuring approximately 12 by 6 meters, 2.5 meters tall. Speakers were on a table approximately three meters from the phones. The room contained office furniture and projection equipment. We measured a background noise level (P_{noise}) of approximately 53 dB.

We tested the commands against two smart phones, a Samsung Galaxy S4 running Android 4.4.2 and Apple iPhone 6 running iOS 9.1 with Google Now app version 9.0.60246. Google’s recently updated [34] default speech recognition system was used to interpret the commands. In the absence of injected ambient background noise, our sound level meter positioned next to the smartphones measured the median intensity of the voice commands to be approximately 88 dB.

We also projected various background noise samples collected from SoundBible [9], recorded from a casino, classroom, shopping mall, and an event during which applause occurred. We varied the volume of these background noises—thus artificially adjusting the signal-to-noise ratio—and played them through eight overhead JBL in-ceiling speakers. We placed a Kinobo “Akiro” table mic next to our test devices and recorded all audio commands that we played to the devices for use in later experiments, described below.

3.4 Evaluation

Attack range. We found that the phone’s speech recognition system failed to identify speech when the speaker was located more than 3.5 meters away or when the perceived SNR was less than 5 dB. We conjecture that the speech recognition system is designed to discard far away noises, and that sound attenuation further limits the attacker’s possible range. While the attacker’s locality is clearly a limitation of this approach, there are many attack vectors that allow the attacker to launch attacks within a few meters of the targeted device, such as obfuscated audio commands embedded in streaming videos, overhead speakers in offices, elevators, or other enclosed spaces, and propagation from other nearby phones.

Machine understanding. Table 2 shows a side-by-side comparison of human and machine understanding, for both normal and obfuscated commands.

The “machine” columns indicate the percentage of trials in which a command is correctly interpreted by the phone, averaged over the various background noises. Here, our sound meter measured the signal’s median audio level at 88 dB and the background noise at 73 dB, corresponding to a signal-to-noise ratio of 15 dB.

Across all three commands, the phones correctly interpreted the normal versions 85% of the time. This accuracy decreased to 60% for obfuscated commands.

We also evaluate how the amplitude of background noise affects machine understanding of the commands. Figure 3 shows the percentage of voice commands that are correctly interpreted by the phones (“success rate”) as a function of the SNR (in dB) using the Mall background noise. Note that a higher SNR denotes more favorable conditions for speech recognition. Generally, Google’s speech recognition engine correctly transcribes the voice commands and activates the phone. The accuracy is higher for normal commands than obfuscated commands, with accuracy improving as SNR increases. In all cases, the speech recognition system is able to perfectly understand and activate the phone functionality in at least some configurations—that is, all of our obfuscated audio commands work at least *some* of the time. With little background noise, the obfuscated commands work extremely well and are often correctly transcribed at least 80% of the time. Appendix B shows detailed results for additional background noises.

Human understanding. To test human understanding of the obfuscated voice commands, we conducted a study on Amazon Mechanical Turk¹, a service that pays

¹**Note on ethics:** Before conducting our Amazon Mechanical Turk experiments, we submitted an online application to our institution’s IRB. The IRB responded by stating that we were exempt from IRB. Irrespective of our IRB, we believe our experiments fall well within the

Table 2: Black-box attack results. The “machine” columns report the percentage of commands that were correctly interpreted by the tested smartphones. The percentage of commands that were correctly understood by humans (Amazon Turk workers) is shown under the “human” columns. For the latter, the authors assessed whether the Turk workers correctly understood the commands.

	Ok Google		Turn on airplane mode		Call 911	
	Machine	Human	Machine	Human	Machine	Human
Normal	90% (36/40)	89% (356/400)	75% (30/40)	69% (315/456)	90% (36/40)	87% (283/324)
Obfuscated	95% (38/40)	22% (86/376)	45% (18/40)	24% (109/444)	40% (16/40)	94% (246/260)

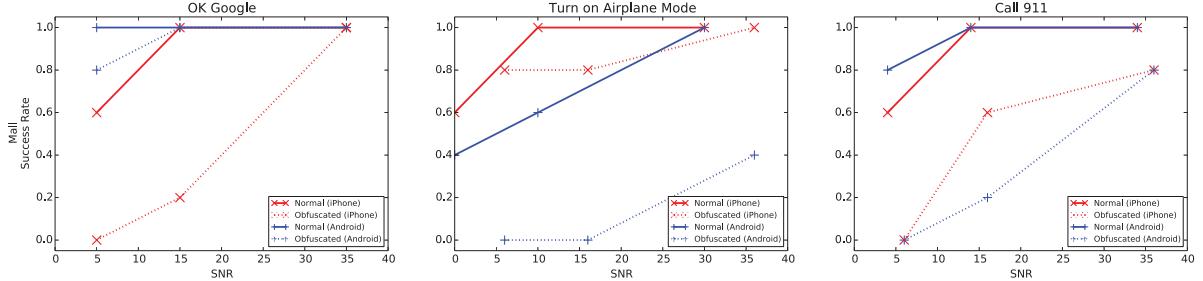


Figure 3: Machine understanding of normal and obfuscated variants of “OK Google”, “Turn on Airplane Mode”, and “Call 911” voice commands under Mall background noise. Each graph shows the measured average success rate (the fraction of correct transcripts) on the y-axis as a function of the signal-to-noise ratio.

human workers to complete online tasks called Human Intelligence Tasks (HITs). Each HIT asks a user to transcribe several audio samples, and presents the following instructions: ‘‘We are conducting an academic study that explores the limits of how well humans can understand obfuscated audio of human speech. The audio files for this task may have been algorithmically modified and may be difficult to understand. Please supply your best guess to what is being said in the recordings.’’

We constructed the online tasks to minimize priming effects—no worker was presented with both the normal and obfuscated variants of the same command. Due to this structuring, the number of completed tasks varies among the commands as reflected in Table 2 under the “human” columns.

We additionally required that workers be over 18 years of age, citizens of the United States, and non-employees of our institution. Mechanical Turk workers were paid \$1.80 for completing a HIT, and awarded an additional \$0.20 for each correct transcription. We could not prevent the workers from replaying the audio samples multiple times on their computers and the workers were incentivized to do so, thus our results could be considered conservative: if the attacks were mounted in practice, device owners might only be able to hear an attack once.

basic principles of ethical research. With respect in particular to beneficence, the Mechanical Turk workers benefited from their involvement (by being compensated). The costs/risks were extremely low: workers were fully informed of their task and no subterfuge occurred. No personal information—either personally identifiable or otherwise—was collected and the audio samples consisted solely of innocuous speech that is very unlikely to offend (e.g., commands such as “OK Google”).

To assess how well the Turk workers understood normal and obfuscated commands, four of the authors compared the workers’ transcriptions to the correct transcriptions (e.g., “OK Google”) and evaluated *whether both had the same meaning*. Our goal was not to assess whether the workers correctly heard the obfuscated command, but more conservatively, whether their perception conformed with the command’s meaning. For example, the transcript “activate airplane functionality” indicates a failed attack even though the transcription differs significantly from the baseline of “turn on airplane mode”.

Values shown under the “human” columns in Table 2 indicate the fraction of total transcriptions for which the survey takers believed that the Turk worker understood the command. Each pair of authors had an agreement of over 95% in their responses, the discrepancies being mainly due to about 5% of responses in which one survey taker believed they matched but the others did not. The survey takers were presented only with the actual phrase and transcribed text, and were blind to whether or not the phrase was an obfuscated command or not.

Turk workers were fairly adept (although not perfect) at transcribing normal audio commands: across all commands, we assessed 81% of the Turkers’ transcripts to convey the same meaning as the actual command.

The workers’ ability to understand obfuscated audio was considerably less: only about 41% of obfuscated commands were labeled as having the same meaning as the actual command. An interesting result is that the black-box attack performed far better for some commands than others. For the “Ok Google” command, we

decreased human transcription accuracy fourfold without any loss in machine understanding.

“Call 911” shows an anomaly: human understanding increases for obfuscated commands. This is due to a tricky part of the black-box attack workflow: the attacker must manage priming effects when choosing an obfuscated command. In this case, we believed the “call 911” candidate command to be unintelligible; these results show we were wrong. A better approach would have been to repeat several rounds of crowdsourcing to identify a candidate that was not understandable; any attacker could do this. It is also possible that among our US reviewers, “call 911” is a common phrase and that they were primed to recognize it outside our study.

Objective measures of human understanding: The analysis above is based on the authors’ assessment of Turk workers’ transcripts. In Appendix C, we present a more objective analysis using the Levenshtein edit distance between the true transcript and the Turkers’ transcripts, with phonemes as the underlying alphabet.

We posit that our (admittedly subjective) assessment is more conservative, as it directly addresses human *understanding* and considers attacks to fail if a human understands the meaning of a command; in contrast, comparing phonemes measures something slightly different—whether a human is able to reconstruct the *sounds* of an obfuscated command—and does not directly capture understanding. Regardless, the phoneme-based results from Appendix C largely agree with those presented above.

4 White-box Attacks

We next consider an attacker who has knowledge of the underlying voice recognition system. To demonstrate this attack, we construct hidden voice commands that are accepted by the open-source CMU Sphinx speech recognition system [24]. CMU Sphinx is used for speech recognition by a number of apps and platforms², making it likely that these whitebox attacks are also practical against these applications.

4.1 Overview of CMU Sphinx

CMU Sphinx uses the Mel-Frequency Cepstrum (MFC) transformation to reduce the audio input to a smaller dimensional space. It then uses a Gaussian Mixture Model (GMM) to compute the probabilities that any given piece of audio corresponds to a given phoneme. Finally, using a Hidden Markov Model (HMM), Sphinx converts the phoneme probabilities to words.

²Systems that use CMU Sphinx speech recognition include the Jasper open-source personal digital assistant and Gnome Desktop voice commands. The Sphinx Project maintains a list of software that uses Sphinx at <http://cmusphinx.sourceforge.net/wiki/sphinxinaction>.

The purpose of the MFC transformation is to take a high-dimensional input space—raw audio samples—and reduce its dimensionality to something which a machine learning algorithm can better handle. This is done in two steps. First, the audio is split into overlapping frames.

Once the audio has been split into frames, we run the MFC transformation on each frame. The Mel-Frequency Cepstrum Coefficients (MFCC) are the 13-dimensional values returned by the MFC transform.

After the MFC is computed, Sphinx performs two further steps. First, Sphinx maintains a running average of each of the 13 coordinates and subtracts off the mean from the current terms. This normalizes for effects such as changes in amplitude or shifts in pitch.

Second, Sphinx numerically estimates the first and second derivatives of this sequence to create a 39-dimensional vector containing the original 13-dimensional vector, the 13-dimensional first-derivative vector, and the 13-dimensional-second derivative vector.

Note on terminology: For ease of exposition and clarity, in the remainder of this section, we call the output of the MFCC function *13-vectors*, and refer to the output after taking derivatives as *39-vectors*.

The Hidden Markov Model. The Sphinx HMM acts on the sequence of 39-vectors from the MFCC. States in the HMM correspond to phonemes, and each 39-vector is assigned a probability of arising from a given phoneme by a Gaussian model, described next. The Sphinx HMM is, in practice, much more intricate: we give the complete description in Appendix A.

The Gaussian Mixture Model. Each HMM state yields some distribution on the 39-vectors that could be emitted while in that state. Sphinx uses a GMM to represent this distribution. The GMMs in Sphinx are a mixture of eight Gaussians, each over \mathbb{R}^{39} . Each Gaussian has a mean and standard deviation over every dimension. The probability of a 39-vector v is the sum of the probabilities from each of the 8 Gaussians, divided by 8. For most cases we can approximate the sum with a maximization, as the Gaussians typically have little overlap.

4.2 Threat model

We assume the attacker has complete knowledge of the algorithms used in the system and can interact with them at will while creating an attack. We also assume the attacker knows the parameters used in each algorithm.³

We use knowledge of the coefficients for each Gaussian in the GMM, including the mean and standard deviation for each dimension and the importance of each

³Papernot et al. [32] demonstrated that it is often possible to transform a white-box attack into a black-box attack by using the black-box as an oracle and reconstructing the model and using the reconstructed parameters.

Gaussian. We also use knowledge of the dictionary file in order to turn words into phonemes. An attacker could reconstruct this file without much effort.

4.3 Simple approach

Given this additional information, a first possible attack would be to use the additional information about exactly what the MFCC coefficients are to re-mount the the previous black-box attack.

Instead of using the MFCC inversion process described in §3.2, this time we implement it using gradient descent—a generic optimization approach for finding a good solution over a given space—an approach which can be generalized to arbitrary objective functions.

Gradient descent attempts to find the minimum (or maximum) value of an objective function over a multi-dimensional space by starting from an initial point and traveling in the direction which reduces the objective most quickly. Formally, given a smooth function f , gradient descent picks an initial point x_0 and then repeatedly improves on it by setting $x_{i+1} = x_i + \epsilon \cdot \nabla f(x_0)$ (for some small ϵ) until we have a solution which is “good enough”.

We define the objective function $f(x) = (\text{MFCC}(x) - y)^2 \cdot z$, where x is the input frame, y is the target MFCC vector, and z is the relative importance of each dimension. Setting $z = (1, 1, \dots, 1)$ takes the L^2 norm as the objective.

Gradient descent is not guaranteed to find the global optimal value. For many problems it finds only a *local* optimum. Indeed, in our experiments we have found that gradient descent only finds local optima, but this turns out to be sufficient for our purposes.

We perform gradient descent search one frame at a time, working our way from the first frame to the last. For the first frame, we allow gradient descent to pick any 410 samples. For subsequent frames, we fix the first 250 samples as the last 250 of the preceding frame, and run gradient descent to find the best 160 samples for the rest of the frame.

As it turns out, when we implement this attack, our results are no better than the previous black-box-only attack. Below we describe our improvements to make attacks completely unrecognizable.

4.4 Improved attack

To construct hidden voice commands that are more difficult for humans to understand, we introduce two refinements. First, rather than targeting a specific sequence of MFCC vectors, we start with the target phrase we wish to produce, derive a sequence of phonemes and thus a sequence of HMM states, and attempt to find an input that matches that sequence of HMM states. This provides

more freedom by allowing the attack to create an input that yields the same sequence of phonemes but generates a different sequence of MFCC vectors.

Second, to make the attacks difficult to understand, we use as few frames per phoneme as possible. In normal human speech, each phoneme might last for a dozen frames or so. We try to generate synthetic speech that uses only four frames per phoneme (a minimum of three is possible—one for each HMM state). The intuition is that the HMM is relatively insensitive to the number of times each HMM state is repeated, but humans are sensitive to it. If Sphinx does not recognize the phrase at the end of this process, we use more frames per phoneme.

For each target HMM state, we pick one Gaussian from that state’s GMM. This gives us a sequence of target Gaussians, each with a mean and standard deviation.

Recall that the MFC transformation as we defined it returns a 13-dimensional vector. However, there is a second step which takes sequential derivatives of 13-vectors to produce 39-vectors. The second step of our attack is to pick these 13-vectors so that after we take the derivatives, we maximize the likelihood score the GMM assigns to the resulting 39-vector. Formally, we wish to find a sequence y_i of 39-dimensional vectors, and x_i of 13-dimensional vectors, satisfying the derivative relation

$$y_i = (x_i, x_{i+2} - x_{i-2}, (x_{i+3} - x_{i-1}) - (x_{i+1} - x_{i-3}))$$

and maximizing the likelihood score

$$\prod_i \exp \left\{ \sum_{j=1}^{39} \frac{\alpha_i^j - (y_i^j - \mu_i^j)^2}{\sigma_i^j} \right\}$$

where μ_i , σ_i , and α_i are the mean, standard deviation, and importance vectors respectively.

We can solve this problem exactly by using the least-squares method. We maximize the *log-likelihood*,

$$\log \prod_i \exp \left\{ \sum_j \frac{-\alpha_i^j + (y_i^j - \mu_i^j)^2}{\sigma_i^j} \right\} = \sum_i \sum_j \frac{-\alpha_i^j + (y_i^j - \mu_i^j)^2}{\sigma_i^j}$$

The log-likelihood is a sum of squares, so maximizing it is a least-squares problem: we have a linear relationship between the x and y values, and the error is a squared difference.

In practice we cannot solve the full least squares problem all at once. The Viterbi algorithm only keeps track of the 100 best paths for each prefix of the input, so if the global optimal path had a prefix that was the 101st most likely path, it would be discarded. Therefore, we work one frame at a time and use the least squares approach to find the next best frame.

This gives us three benefits: First, it ensures that at every point in time, the next frame is the best possible given what we have done so far. Second, it allows us to

try all eight possible Gaussians in the GMM to pick the one which provides the highest score. Third, it makes our approach more resilient to failures of gradient descent. Sometimes gradient descent cannot hit the 13-vector suggested by this method exactly. When this happens, the error score for subsequent frames is based on the actual 13-vector obtained by gradient descent.

Complete description. We first define two subroutines to help specify our attack more precisely. $\text{LSTDERRIV}(f, \bar{g}, g)$ accepts a sequence of 13-vectors f that have already been reached by previous iterations of search, the desired 39-vector sequence \bar{g} , and one new 39-vector g ; it uses least squares to compute the next 13-vector which should be targeted along with the least-squares error score. Specifically:

1. Define A as the $39k \times 13(6+k)$ dimensional matrix which computes the derivative of a sequence of $6+k$ 13-vectors and returns the k resulting 39-vectors.
2. Define b as the $39k$ dimensional vector corresponding to the concatenation of the $k-1$ 39-vectors in \bar{g} and the single 39-vector g .
3. Split A in two pieces, with A_L being the left $13k$ columns, and A_R being the right 6×13 columns.
4. Define \bar{f} as the concatenation of the 13-vectors in f .
5. Define $\bar{b} = b - A_L \cdot \bar{f}$.
6. Using least squares, find the best approximate solution \hat{x} to the system of equations $A_R \cdot \hat{x} = \bar{b}$.
7. Return $(|(A_R \cdot \hat{x}) - \bar{b}|, \hat{x})$

$\text{GRADDESC}(s, t)$ accepts the previous frame $s \in \mathbb{R}^{410}$ and a target 13-vector t , and returns a frame $\hat{s} \in \mathbb{R}^{410}$ such that \hat{s} matches s in the 250 entries where they overlap and $\text{MFCC}(\hat{s})$ is as close to t as possible. More precisely, it looks for a 160-dimensional vector x that minimizes $f(x) = \|\text{MFCC}(s_{160..410}||x) - s\|_2$, where $||$ is concatenation, and returns $s_{160..410}||x$. We use the Newton Conjugate-Gradient algorithm for gradient descent and compute the derivative symbolically for efficiency.

Our full algorithm works as follows:

1. In the following, f will represent a sequence of chosen 13-vectors (initially empty), \bar{g} a sequence of target 39-vectors, s the audio samples to return, and i the iteration number (initially 0).
2. Given the target phrase, pick HMM states h_i such that each state corresponds to a portion of a phoneme of a word in the phrase.
3. Let g_i^j be the 39-vector corresponding to the mean of the j^{th} Gaussian of the GMM for this HMM state. One of these will be the target vector we will try to invert.
4. For each j , solve the least squares problem $(s_j, d_j) = \text{LSTDERRIV}(f, \bar{g}, g_i^j)$ and set $\hat{j} = \arg \min_j s_j$ and $\bar{d} = d_{\hat{j}}$ to obtain a sequence of 13-vectors \bar{d}_0 to \bar{d}_{i+6} . Let \bar{d}_i be the “target 13-vector”

5. Append the 39-vector corresponding to t to \bar{g} .
6. Use gradient descent to get $\hat{s} = \text{GRADDESC}(s, t)$. Let $s := \hat{s}$. Append $\text{MFCC}(s)$ to f .
7. Repeat for the next i from step 3 until all states are completed.

4.5 Playing over the air

The previous attacks work well when we feed the audio file directly into Sphinx. However, Sphinx could not correctly transcribe recordings made by playing the audio using speakers. We developed three approaches to solve this complication:

Make the audio easier to play over speaker. Gradient descent often generates audio with very large spikes. It’s physically impossible for the speaker membrane to move quickly enough to accurately reproduce these spikes. We modified gradient descent to penalize waveforms that a speaker cannot reproduce. In particular, we add a penalty for large second derivatives in the signal, with the hope that gradient descent finds solutions that do not include such large spikes.

Predict the MFCC of played audio. Even with this penalty, the audio is still not perfectly playable over a speaker. When we compared the waveform of the played audio and recorded audio, they had significant differences. To address this, we built a model to predict the MFCC when a file is played through a speaker and recorded. Recall that the MFCC transformation essentially computes the function $C \log(B \|Ax\|^2)$.

By playing and recording many audio signals, we learned new matrices \hat{A} , \hat{B} , \hat{C} so that for each played frame x and recorded frame y , $C \log(B \|Ay\|^2)$ is close to $\hat{C} \log(\hat{B} \|\hat{A}x\|^2)$. We computed \hat{A} , \hat{B} , \hat{C} by solving a least-squares problem. This was still not enough for correct audio recognition, but it did point us in a promising direction.

Play the audio during gradient descent. The ideas above are not enough for recognition of recorded audio. To see what is going on here, we compare the MFCC of the played audio (after recording it) and the initial audio (before playing it). We found the correlation to be very high ($r = .97$ for the important coefficients).

Based on this observation, we augment our algorithm to include an outer iteration of gradient descent. Given a target MFCC we first run our previous gradient descent algorithm to find a sound sequence which (before playing over the speaker) reaches the target MFCC. Then, we play and record it over the speaker. We obtain from this the actual MFCC. We then adjust the target MFCC by the difference between what was received and what is desired.

We implemented this approach. Figure 4 plots the L^2

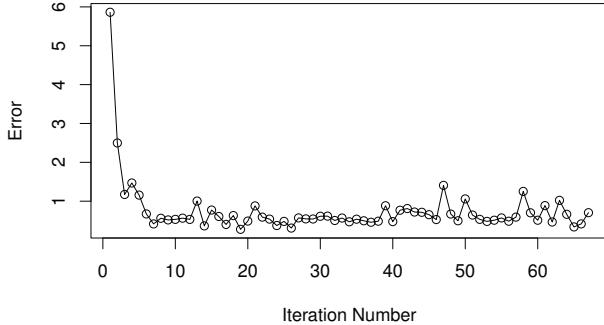


Figure 4: Incorporating actually playing the audio over the speakers into the gradient descent significantly reduces the error. The plot is of the L^2 norm of the error of from the target feature vector to the actually recorded feature vector, over time.

error (the difference between the target MFCC and what is actually recorded during each iteration of our algorithm) over time. By repeating this procedure 50 times and taking the frame with the minimum noise, we obtain an audio file that is correctly recognized by Sphinx after being played over the speaker.

Since we perform 50 iterations of the inner gradient descent per frame, and each iteration takes 30 seconds, our approach takes nearly 30 hours to find a valid attack sample. In practice, sometimes this process can take even longer; since we are recording audio, if the microphone picks up too much background noise, we must discard the recorded sample and try again. We have built in an error-detection system to mitigate these effects.

This might seem like a high cost to generate one attack sample. However, once generated, we can reuse the obfuscated audio on that speaker forever. Even though the setup cost is high, it must only be performed once; thereafter the same audio file can be used repeatedly.

4.6 Evaluation

Machine comprehension. For the former, we apply the above techniques and generate three audio commands: “okay google, take a picture”, “okay google, text 12345”, and “okay google, browse to evil.com”. The speech recognition system is an instance of CMU Sphinx version 4-1.0beta6.

We determined the minimum number of frames per phoneme that is sufficient to allow Sphinx to recognize the command. Some words are more difficult to create correctly than others, and thus require more frames per phoneme. Detailed results can be found in Appendix E. When we modify the lengths of the phonemes to account for this data, *over 90% of generated phrases are correctly recognized by Sphinx*.

To evaluate our attack playing over a microphone, we equipped our computer with an external Blue Snowball

Table 3: White-box attack results. Percentages show successful comprehension of Normal and Obfuscated version of voice commands for humans and machines. Our white-box attack constructs the obfuscated attack without a starting normal attack; we only evaluate the machine understanding of the obfuscated commands.

	Human Understanding	Machine Understanding
Normal	74% (230/310)	–
Obfuscated	0% (1/377)	82% (82/100)

microphone. Obfuscated audio commands are played from an Altec Lansing ASC45.1 speaker located 0.5 meters from the microphone. All of our evaluated phrases were commands recorded from the microphone. We trained our MFCC vector predictor with one minute (16 million samples) of audio played over this speaker in a large room to minimize echoes. The room was isolated, and thus we encountered nearly no background noise. The attack took 32 hours to generate the command for “okay google, take a picture”, but required no human intervention, using the same number of frames chosen from the previous attack for a total of 101 frames.

Once the file was generated, we played the command over the speaker 100 times. Sphinx was able to recognize it in 82 of the instances. This is to be expected—for a given audio played multiple times, there is a very high (99%+) correlation between the recorded MFCC values.

Human comprehension. As with our black-box attack, we evaluate our obfuscated commands with the same Amazon Mechanical Turk study. Workers were blind to how the obfuscated audio was generated. As a baseline, we asked workers to transcribe a phrase spoken by a human naturally. The results show that our white-box attack is significantly better than the black-box attack. No worker was able to transcribe more than half of the words in the phrase correctly, and 75% of workers transcriptions shared no words in common with the actual phrase.

Four of the authors independently evaluated 377 transcriptions of attack commands generated by our white-box attack and compared them with correct transcriptions to answer whether both transcriptions had the same meaning. The four authors agreed on over 95% of instances. Table 3 contains the results, showing that normal commands were correctly transcribed by workers 74% of the time, but obfuscated commands were almost never transcribed correctly. A comparison using phoneme-level edit distance yields similar results; see Appendix C.2.

While these results indicate that obfuscated commands generated using our white-box attack are very difficult to understand, we conducted a second study to determine

if users actually thought the audio was human speech or just noise. Specifically, we created audio samples of a human speaking a phrase, followed by an obfuscated (different) phrase, and finally a human speaking a third different phrase. In this study we were interested in seeing if the worker would try to transcribe the obfuscated speech at all, or leave it out entirely.

Transcription accuracy was 80% for the first and last commands given by a human speaking. Only 24% of users attempted to transcribe the obfuscated speech. This study clearly demonstrates that when given a choice about what they viewed as speech and not-speech, the majority of workers believed our audio was not speech.

5 Defenses

We are unaware of any device or system that currently defends against obfuscated voice commands. In this section, we explore potential defenses for hidden voice commands across three dimensions: *defenses that notify*, *defenses that challenge*, and *defenses that detect and prohibit*. The defenses described below are not intended to be exhaustive; they represent a first examination of potential defenses against this new threat.

5.1 Defenses that notify

As a first-line of defense we consider defenses that alert the user when the device interprets voice commands, though these will only be effective when the device operator is present and notification is useful (e.g., when it is possible to undo any performed action).

The “Beep”, the “Buzz” and the “Lightshow”. These defenses are very simple: when the device receives a voice command, it notifies the user, e.g., by beeping. The goal is to make the user aware a voice command was accepted. There are two main potential issues with “the Beep”: (i) attackers may be able to mask the beep, or (ii) users may become accustomed to their device’s beep and begin to ignore it. To mask the beep, the attacker might play a loud noise concurrent with the beep. This may not be physically possible depending on the attacker’s speakers and may not be sufficiently stealthy depending on the environment as the noise require can be startling.

A more subtle attack technique is to attempt to mask the beep via noise cancellation. If the beep were a single-frequency sine wave an attacker might be able to cause the user to hear nothing by playing an identical frequency sine wave that is out of phase by exactly half a wavelength. We evaluated the efficacy of this attack by constructing a mathematical model that dramatically oversimplifies the attacker’s job and shows that even this simplified “anti-beep” attack is nearly impossible. We

present a more detailed evaluation of beep cancellation in Appendix D.

Some devices might inform the user when they interpret voice commands by vibrating (“the buzz”) or by flashing LED indicators (“the lightshow”). These notifications also assume that the user will understand and heed such warnings and will not grow accustomed to them. To differentiate these alerts from other vibration and LED alerts the device could employ different pulsing patterns for each message type. A benefit of such notification techniques is that they have low overhead: voice commands are relatively rare and hence generating a momentary tone, vibration, or flashing light consumes little power and is arguably non-intrusive.

Unfortunately, users notoriously ignore security warning messages, as is demonstrated by numerous studies of the (in)effectiveness of warning messages in deployed systems [35, 37, 44]. There is unfortunately little reason to believe that most users would recognize and not quickly become acclimated to voice command notifications. Still, given the low cost of deploying a notification system, it may be worth considering *in combination* with some of the other defenses described below.

5.2 Defenses that challenge

There are many ways in which a device may seek confirmation from the user before executing a voice command. Devices with a screen might present a confirmation dialogue, though this limits the utility of the voice interface. We therefore consider defenses in which the user must vocally confirm interpreted voice commands. Presenting an audio challenge has the advantage of requiring the user’s attention, and thus may prevent all hidden voice commands from affected the device assuming the user will not confirm an unintended command. A consistent verbal confirmation command, however, offers little protection from hidden voice commands: the attacker also provide the response in an obfuscated manner. If the attacker can monitor any random challenge provided by the device, it might also be spoofed. To be effective, the confirmation must be easily produced by the human operator and be difficult to forge by an adversary.

The Audio CAPTCHA. Such a confirmation system already exists in the form of audio CAPTCHAs [26] which is a challenge-response protocol in which the challenge consists of speech that is constructed to be difficult for computers to recognize while being easily understood by humans. The response portion of the protocol varies by the type of CAPTCHA, but commonly requires the human to transcribe the challenge.

Audio CAPTCHAs present an possible defense to hidden voice commands: before accepting a voice command, a device would require the user to correctly re-

spond to an audio CAPTCHA, something an attacker using machine speech recognition would find difficult. While it is clear that such a defense potentially has usability issues, it may be worthwhile for commands that are damaging or difficult to undo.

Audio CAPTCHAs are useful defenses against hidden voice commands only if they are indeed secure. Previous generations of audio CAPTCHAs have been shown to be broken using automated techniques [6, 39]. As audio CAPTCHAs have improved over time [11, 27], the question arises if currently fielded audio CAPTCHAs have kept pace with improvements in speech recognition technologies. In short, they have not.

We focus our examination on two popular audio CAPTCHA systems: Google’s *reCaptcha* [33] offers audio challenges initially consisting of five random digits spread over approximately ten seconds; and *NLP Captcha* [30] provides audio challenges of about three seconds each composed of four or five alphanumeric characters, with the addition of the word “and” before the last character in some challenges.

We tested 50 challenges of *reCaptcha* and *NLP Captcha* each by segmenting the audio challenges before transcribing them using Google’s speech recognition service. Figure 5 shows the results of transcription. Here, we show the normalized edit distance, which is the Levenshtein edit distance using characters as alphabet symbols divided by the length of the challenge. More than half and more than two-thirds of NLP Captchas and *reCaptcha*, respectively, are perfectly transcribed using automated techniques. Moreover, approximately 80% of CAPTCHAs produced by either system have a normalized edit distance of 0.3 or less, indicating a high frequency of at least mostly correct interpretations. This is relevant, since audio CAPTCHAs are unfortunately not easily understood by humans; to increase usability, *reCaptcha* provides some “leeway” and accepts almost-correct answers.

Given the ease at which they can be solved using automated techniques, the current generation of deployed audio CAPTCHA systems seems unsuitable for defending against hidden voice commands. Our results do not indicate whether or not audio CAPTCHAs are necessarily insecure. However, we remark that since computers continue to get better at speech recognition developing robust audio CAPTCHA puzzles is likely to become increasingly more difficult.

5.3 Defenses that detect and prevent

Speaker recognition. Speaker recognition (sometimes called voice authentication) has been well-explored as a biometric for authentication [7], with at least Google recently including speaker recognition as

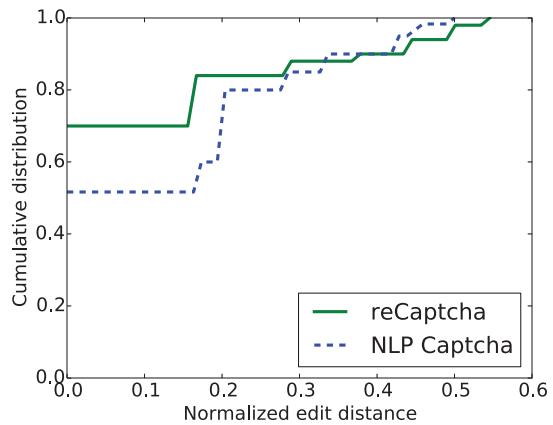


Figure 5: Accuracy of breaking audio CAPTCHA using machine based speech-to-text conversion. A normalized edit distance of zero signifies exact prediction.

an optional feature in its Android platform [18]. Apple also introduced similar functionality in iOS [2].

However, it is unclear whether speaker verification necessarily prevents the use of hidden voice commands, especially in settings in which the adversary may be able to acquire samples of the user’s voice. Existing work has demonstrated that voices may be mimicked using statistical properties⁴; for example, Aylett and Yamagishi [3] are able to mimic President George W. Bush’s voice with as little of 10 minutes of his speech. Hence, it may be possible to construct an obfuscated voice command based on recordings of the user’s voice that will be accepted both by the speaker recognition and the voice recognition systems. This is an interesting technical question which we defer to future work.

Importantly, speaker recognition presents three well-understood usability issues. First, a non-negligible false negative rate might limit authorized use, which provides an incentive for users to deactivate speaker recognition. Second, speaker recognition requires training, and likely necessitates the collection of a large speech corpus in order to ensure the level of accuracy necessary for reliable authentication. This need is compounded for devices such as Amazon Echo that are intended to be used by multiple users. Users may be unwilling to perform such training. Finally, speaker recognition is unsuited for applications that have no prior interactions with the user, for example, in kiosks for the visually impaired. We argue that such usability issues motivate the need for less intrusive defenses, such as the ones described next.

The “Filter”. This defense decreases the fidelity of the input audio before applying speech recognition. This

⁴CereVoice offers an online service for “[creating] a computer version of your own voice” [8].

somewhat counterintuitive approach leverages the precision required by hidden voice commands: by slightly degrading the audio quality, normal commands are affected only slightly while obfuscated inputs, already at the cusp of being comprehensible by machine, are no longer recognized. The goal of the filter is thus to find a “sweet spot” such that a slight decrease in fidelity will not too adversely affect normal audio, but will eliminate hidden voice commands.

Our filter implementation considers audio in the time domain. For a filter rate f and an audio file consisting of s samples, we preserve $f \cdot s$ samples chosen uniformly at random and remove without replacement the remaining $(1 - f)s$ samples.

We evaluate the efficacy of the filter defense using the West Point Company English speech data corpus from the University of Pennsylvania’s Linguistic Data Consortium (LDC) [29]. The LDC corpus consists of American English-language speakers, 30 of them speaking the same 50 sentences each, and another 109 speakers each speaking a separate set of 185 sentences. For each sentence in the corpus, we construct multiple obfuscated audio files using the black-box techniques described in §3. We do not introduce any background noise to either the normal or obfuscated samples, and send all samples to Google’s speech recognition system and evaluate the accuracy of its transcriptions. We tested the filter against 16,440 normal commands and 27,589 obfuscated commands, sampled uniformly at random without replacement. The audio files were in WAV format with single channel sampled at 22050 kHz with 32-bit bit depth in float data type.

We define *phoneme edit distance* δ as the Levenshtein edit distance between phoneme sequences of two transcriptions, and L as the phoneme length of unfiltered sentence. Finally we define *relative phoneme edit distance* as $\delta / (\delta + L)$, to reflect how close the two transcriptions might sound to a human listener. Figure 6 plots the cumulative distribution of the *relative phoneme edit distances* for our tested audio samples. Out of five transcripts Google returned, we pick the transcript with the smallest phoneme edit distance and calculate the *relative phoneme edit distance*. With an 80% filter (i.e., 80% of the original audio is preserved), the hidden voice command is significantly perturbed—the fraction of correctly transcribed sentences decreases from 41.74% to 7.49% when the filter is applied. However, the filter also significantly degrades normal audio, decreasing accuracy by 14.35%, which is likely unacceptable for users who depend on voice functionality. Through manual tuning, we find that a 92% filter offers an attractive sweet spot, causing the fraction of correctly transcribed obfuscated files to fall from 41.74% to 26.60%, while negligibly affecting normal files, whose accuracy drops only 1.06%.

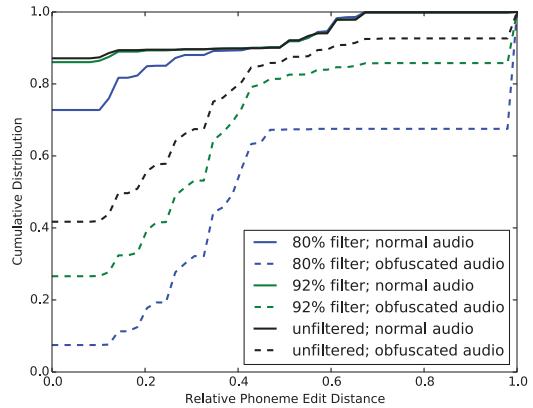


Figure 6: The cumulative distribution of relative phoneme edit distances for normal and obfuscated commands, for filtering rates of 80% and 92%.

The “Classifier”. As a final defense we examine whether automated techniques can detect our hidden voice commands—that is, can a computer classify interpreted speech as being generated by a human or a machine? In practice, a speech recognition system could incorporate a machine learning classifier as a final processing step, dropping voice commands that are determined to be machine-generated.

We construct a hidden voice command detector using logistic regression. Our classifier uses mid-term features (mean and standard deviation) derived from short term features shown in Appendix F, extracted using the pyAudioAnalysis library [16] from the original audio signal.

We use the same LDC speech corpus as used in our filter experiments, and label each sample from the corpus as *normal*. Using the black-box procedure described in §3, we produce obfuscated hidden voice commands, which we label as *obfuscated*. We overlay the background noise samples described in §3.3 to mimic a noisy environment to both the *normal* and *obfuscated* samples.

We apply tenfold cross validation and report averages across the ten runs. The confusion matrix that results from classification is reported in Table 4. 99.8% of obfuscated audio were detected as such, while only incurring a 0.2% false positive rate, meaning that the classifier would incorrectly discard two out of 1000 valid commands.

To better gauge the classifier’s accuracy when presented with diverse inputs, we performed additional classification tests against 51 commands generated using the white-box technique from §4 and audio from the Accent GMU dataset [43]. The Accent GMU dataset is comprised of 569 audio samples of English text spoken by different individuals with different regional accents. Neither the GMU or white-box samples were used to con-

Table 4: Confusion matrix of our classifier.

	Normal	Obfuscated
Normal	49.9%	0.1%
Obfuscated	0.1%	49.9%

Table 5: Confusion matrix of our classifier, when classifying audio from outside corpora.

	Normal	Attack
White-box Attack	30.7%	69.3%
Accent GMU	99.2%	0.8%

struct the classifier. That is, our results show the efficacy of a classifier constructed with only normal and black-box obfuscated command samples as training data. Importantly, the GMU dataset consists of all normal (non-obfuscated) samples, while the white-box dataset contains only attack commands. The confusion matrix for this classification task is presented in Table 5. For the GMU dataset, our classifier performs well, incurring less than a 1% false positive rate. The performance is worse for the white-box attack. Still, even against this strong attack which requires complete knowledge of the backend speech recognition system, the classifier is able to flag nearly 70% of the hidden voice commands as being malicious.

5.4 Summary of defenses

We present the first examination of defenses against hidden voice commands. Our analysis of notification defenses (§5.1) shows that security alerts are difficult to mask, but may be ignored by users. Still, given their ease of deployment and small footprint, such defenses are worth considering. Active defenses, such as audio CAPTCHAs (§5.2) have the advantage that they require users to affirm voice commands before they become effected. Unfortunately, active defenses also incur large usability costs, and the current generation of audio-based reverse Turing tests seem easily defeatable. Most promising are prevention and detection defenses (§5.3). Our findings show that filters which slightly degrade audio quality can be tuned to permit normal audio while effectively eliminating hidden voice commands. Likewise, our initial exploration of machine learning-based defenses shows that simple classification techniques yield high accuracy in distinguishing between user- and computer-generated voice commands.

6 Limitations and Discussion

While the results of our defenses are encouraging, a limitation of this paper is that the defenses do not offer proofs of security. In particular, an adversary may be able to

construct hidden voice commands that are engineered to withstand filtering and defeat classifiers.

The random sampling used by our filter complicates the task of designing a “filter-resistant” hidden voice command since the adversary has no advanced knowledge of what components of his audio command will be discarded. The adversary is similarly constrained by the classifier, since the attacks we describe in §3 and §4 significantly affect the features used in classification. Of course, there might be other ways to conceal voice commands that are more resistant to information loss yet retain many characteristics of normal speech, which would likely defeat our existing detection techniques. Designing such attacks is left as a future research direction.

The attacks and accompanying evaluations in §3 and §4 demonstrate that hidden voice commands are effective against modern voice recognition systems. There is clearly room for another security arms race between more clever hidden voice commands and more robust defenses. We posit that, unfortunately, the adversary will likely always maintain an advantage so long as humans and machines process speech dissimilarly. That is, there will likely always be some room in this asymmetry for “speaking directly” to a computational speech recognition system in a manner that is not human parseable.

Future work. CMU Sphinx is a “traditional” approach to speech recognition which uses a hidden Markov model. More sophisticated techniques have recently begun to use neural networks. One natural extension of this work is to extend our white-box attack techniques to apply to RNNs.

Additional work can potentially make the audio even more difficult for a human to detect. Currently, the white-box hidden voice commands sound similar to white noise. An open question is if it might be possible to construct working attacks that sound like music or other benign noise.

7 Conclusion

While ubiquitous voice-recognition brings many benefits its security implications are not well studied. We investigate hidden voice commands which allow attackers to issue commands to devices which are otherwise unintelligible to users.

Our attacks demonstrate that these attacks are possible against currently-deployed systems, and that when knowledge of the speech recognition model is assumed more sophisticated attacks are possible which become much more difficult for humans to understand. (Audio files corresponding to our attacks are available at <http://hiddenvoicecommands.com>.)

These attacks can be mitigated through a number of different defenses. Passive defenses that notify the user

an action has been taken are easy to deploy and hard to stop but users may miss or ignore them. Active defenses may challenge the user to verify it is the owner who issued the command but reduce the ease of use of the system. Finally, speech recognition may be augmented to detect the differences between real human speech and synthesized obfuscated speech.

We believe this is an important new direction for future research, and hope that others will extend our analysis of potential defenses to create sound defenses which allow for devices to securely use voice-commands.

Acknowledgments. We thank the anonymous reviewers for their insightful comments. This paper is partially funded from National Science Foundation grants CNS-1445967, CNS-1514457, CNS-1149832, CNS-1453392, CNS-1513734, and CNS-1527401. This research was additionally supported by Intel through the ISTC for Secure Computing, and by the AFOSR under MURI award FA9550-12-1-0040. The findings and opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

- [1] I. Androulopoulos, J. Koutsias, K. V. Chandrinos, and C. D. Spyropoulos. An Experimental Comparison of Naive Bayesian and Keyword-based Anti-spam Filtering with Personal e-Mail Messages. In *ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2000.
- [2] Apple. Use Siri on your iPhone, iPad, or iPod touch. Support article. Available at <https://support.apple.com/en-us/HT204389>.
- [3] M. P. Aylett and J. Yamagishi. Combining Statistical Parametric Speech Synthesis and Unit-Selection for Automatic Voice Cloning. In *LangTech*, 2008.
- [4] M. Barreno, B. Nelson, A. D. Joseph, and J. Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010.
- [5] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion Attacks against Machine Learning at Test Time. In *Machine Learning and Knowledge Discovery in Databases*, 2013.
- [6] E. Bursztein and S. Bethard. Decaptcha: Breaking 75% of eBay Audio CAPTCHAs. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.
- [7] J. Campbell, J.P. Speaker Recognition: A Tutorial. *Proceedings of the IEEE*, 85(9):1437–1462, 1997.
- [8] CereVoice Me Voice Cloning Service. <https://www.cereproc.com/en/products/cerevoiceme>.
- [9] Crowd Sounds — Free Sounds at SoundBible. <http://soundbible.com/tags-crowd.html>.
- [10] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma. Adversarial Classification. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2004.
- [11] M. Darnstadt, H. Meutznar, and D. Kolossa. Reducing the Cost of Breaking Audio CAPTCHAs by Active and Semi-supervised Learning. In *International Conference on Machine Learning and Applications (ICMLA)*, 2014.
- [12] W. Diao, X. Liu, Z. Zhou, and K. Zhang. Your Voice Assistant is Mine: How to Abuse Speakers to Steal Information and Control Your Phone. In *ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, 2014.
- [13] H. Drucker, S. Wu, and V. Vapnik. Support vector machines for spam categorization. *IEEE Transactions on Neural Networks*, 10(5), Sep 1999.
- [14] A. Fawzi, O. Fawzi, and P. Frossard. Analysis of classifiers’ robustness to adversarial perturbations. *arXiv preprint arXiv:1502.02590*, 2015.
- [15] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, 2015.
- [16] T. Giannakopoulos. Python Audio Analysis Library: Feature Extraction, Classification, Segmentation and Applications. <https://github.com/tyiannak/pyAudioAnalysis>.
- [17] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
- [18] Google. Turn on “Ok Google” on your Android. Support article. Available at <https://support.google.com/websearch/answer/6031948>.
- [19] Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images, 2015. IEEE.
- [20] C. Ittiachareon, S. Suksri, and T. Yingthawornsuks. Speech recognition using MFCC. In *International Conference on Computer Graphics, Simulation and Modeling (ICGSM)*, 2012.
- [21] Y. Jang, C. Song, S. P. Chung, T. Wang, and W. Lee. A11y Attacks: Exploiting Accessibility in Operating Systems. In *ACM Conference on Computer and Communications Security (CCS)*, November 2014.
- [22] A. Kantchelian, S. Afroz, L. Huang, A. C. Islam, B. Miller, M. C. Tschantz, R. Greenstadt, A. D. Joseph, and J. D. Tygar. Approaches to Adversarial Drift. In *ACM Workshop on Artificial Intelligence and Security*, 2013.
- [23] C. Kasmi and J. Lopes Esteves. Iemi threats for information security: Remote command injection on modern smartphones. *IEEE Transactions on Electromagnetic Compatibility*, PP(99):1–4, 2015.
- [24] P. Lamere, P. Kwok, W. Walker, E. Gouveia, R. Singh, B. Raj, and P. Wolf. Design of the CMU Sphinx-4 Decoder. In *Eighth European Conference on Speech Communication and Technology*, 2003.
- [25] A. Mahendran and A. Vedaldi. Understanding deep image representations by inverting them. In *Conference on Computer Vision and Pattern Recognition (CVPR) 2015*, 2015.
- [26] M. May. Inaccessibility of CAPTCHA: Alternatives to Visual Turing Tests on the Web. Technical report, W3C Working Group Note, 2005. Available at <http://www.w3.org/TR/turingtest/>.
- [27] H. Meutznar, S. Gupta, and D. Kolossa. Constructing Secure Audio CAPTCHAs by Exploiting Differences Between Humans and Machines. In *Annual ACM Conference on Human Factors in Computing Systems (CHI)*, 2015.
- [28] D. E. Meyer and R. W. Schvaneveldt. Facilitation in Recognizing Pairs of Words: Evidence of a Dependence between Retrieval Operations. *Journal of Experimental Psychology*, 90(2):227, 1971.
- [29] J. Morgan, S. LaRocca, S. Bellinger, and C. C. Ruscelli. West Point Company G3 American English Speech. Linguistic Data Consortium, item LDC2005S30. University of Pennsylvania. Available at <https://catalog.ldc.upenn.edu/LDC2005S30>, 2005.
- [30] NLP Captcha. <http://nlpcaptcha.in/>.
- [31] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. *arXiv preprint arXiv:1511.07528*, 2015.
- [32] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.
- [33] reCAPTCHA. <http://google.com/recaptcha>.
- [34] H. Sak, A. Senior, K. Rao, F. Beaufays, and J. Schalkwyk. Google Voice Search: Faster and More Accurate, 2015. Google Research Blog post. Available at <http://googleresearch.blogspot.com/2015/09/google-voice-search-faster-and-more.html>.
- [35] S. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor’s New Security Indicators: An Evaluation of Website Authentication and the Effect of Role Playing on Usability Studies. In *IEEE Symposium on Security and Privacy (Oakland)*, 2007.
- [36] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2011.
- [37] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *USENIX Security Symposium (USENIX)*, 2009.
- [38] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks. *arXiv preprint arXiv:1312.6199*, 2013.
- [39] J. Tam, J. Simsa, S. Hyde, and L. V. Ahn. Breaking Audio CAPTCHAs. In *Advances in Neural Information Processing Systems (NIPS)*, 2008.
- [40] J. Tygar. Adversarial Machine Learning. *IEEE Internet Computing*, 15(5):4–6, 2011.
- [41] T. Vaidya, Y. Zhang, M. Sherr, and C. Shields. Cocaine Noodles: Exploiting the Gap between Human and Machine Speech Recognition. In *USENIX Workshop on Offensive Technologies (WOOT)*, August 2015.
- [42] O. Viikki and K. Laurila. Cepstral Domain Segmental Feature Vector Normalization for Noise Robust Speech Recognition. *Speech Communication*, 25(13):133–147, 1998.
- [43] S. H. Weinberger. Speech Accent Archive. George Mason University, 2015. Available at <http://accent.gmu.edu>.
- [44] M. Wu, R. C. Miller, and S. L. Garfinkel. Do Security Toolbars Actually Prevent Phishing Attacks? In *SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2006.

A Additional Background on Sphinx

As mentioned in §4, the first transform taken by Sphinx is to split the audio in to overlapping frames, as shown in Figure 7. In Sphinx, frames are 26ms (410 samples) long, and a new frame begins every 10ms (160 samples).

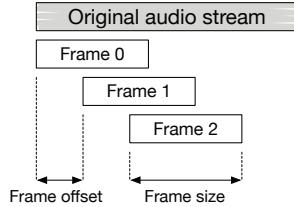


Figure 7: The audio file is split into overlapping frames.

MFC transform. Once Sphinx creates frames, it runs the MFC algorithm. Sphinx’s MFC implementation involves five steps:

1. **Pre-emphasizer:** Applies a high-pass filter that reduces the amplitude of low-frequencies.
2. **Cosine windower:** Weights the samples of the frame so the earlier and later samples have lower amplitude.
3. **FFT:** Computes the first 257 terms of the (complex-valued) Fast Fourier Transform of the signal and returns the squared norm of each.
4. **Mel filter:** Reduces the dimensionality further by splitting the 257 FFT terms into 40 buckets, summing the values in each bucket, then returning the log of each sum.
5. **DCT:** Computes the first 13 terms of the Discrete Cosine Transform (DCT) of the 40 bucketed values.⁵

Despite the many steps involved in the MFC pipeline, the entire process (except the running average and derivatives steps) can be simplified into a single equation:

$$\text{MFCC}(x) = C \log(B \|Ax\|^2)$$

where the norm, squaring and log are done component-wise to each element of the vector. A is a 410×257 matrix which contains the computation performed by the pre-emphasizer, cosine windower, and FFT. B is a 257×40 matrix which computes the Mel filter, and C is a 40×13 matrix which computes the DCT.

Sphinx is configured with a dictionary file, which lists all valid words and maps each word to its phonemes,

⁵While it may seem strange to take the DCT of the frequency-domain data, this second FFT is able to extract higher-level features about which frequencies are common, and is more tolerant to a change in pitch.

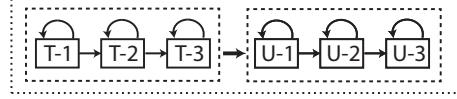


Figure 8: The HMM used by Sphinx encoding the word “two”. Each phoneme is split into three HMM states (which may repeat). These HMM states must occur in sequence to complete a phoneme. The innermost boxes are the phoneme HMM states; the two dashed boxes represent the phoneme, and the outer dashed box the word “two”.

and a grammar file, which specifies a BNF-style formal grammar of what constitutes a valid sequence of words. In our experiments we omit the grammar file and assume any word can follow any other with equal probability. (This makes our job as an attacker more difficult.)

The HMM states can be thought of as phonemes, with an edge between two phonemes that can occur consecutively in some word. Sphinx’s model imposes additional restrictions: its HMM is constructed so that all paths in the HMM correspond to a valid sequence of words in the dictionary. Because of this, any valid path through the HMM corresponds to a valid sequence of words. For example, since the phoneme “g” never follows itself, the HMM only allows one “g” to follow another if they are the start and end of words, respectively.

The above description is slightly incomplete. In reality, each phoneme is split into three HMM states, which must occur in a specific order, as shown in Figure 8. Each state corresponds to the beginning, middle, or end of a phoneme. A beginning-state has an edge to the middle-state, and the middle-state has an edge to the end-state. The end-phoneme HMM state connects to beginning-phoneme HMM states of other phonemes. Each state also has a self-loop that allows the state to be repeated.

Given a sequence of 39-vectors, Sphinx uses the Viterbi algorithm to try to find the 100 most likely paths through the HMM model (or an approximation thereto).

B Detailed Machine Comprehension of Black-box Attack

The detailed results of machine comprehension of black-box attacks are presented in Figure 9.

We note that Figure 9 contains an oddity: in a few instances, the transcription success rate decreases as the SNR increases. We suspect that this is due to our use of median SNR, since the background samples contain non-uniform noise and transient spikes in ambient noise levels may adversely affect recognition. Overall, however, we observe a clear (and expected) trend in which transcription accuracy improves as SNR increases.

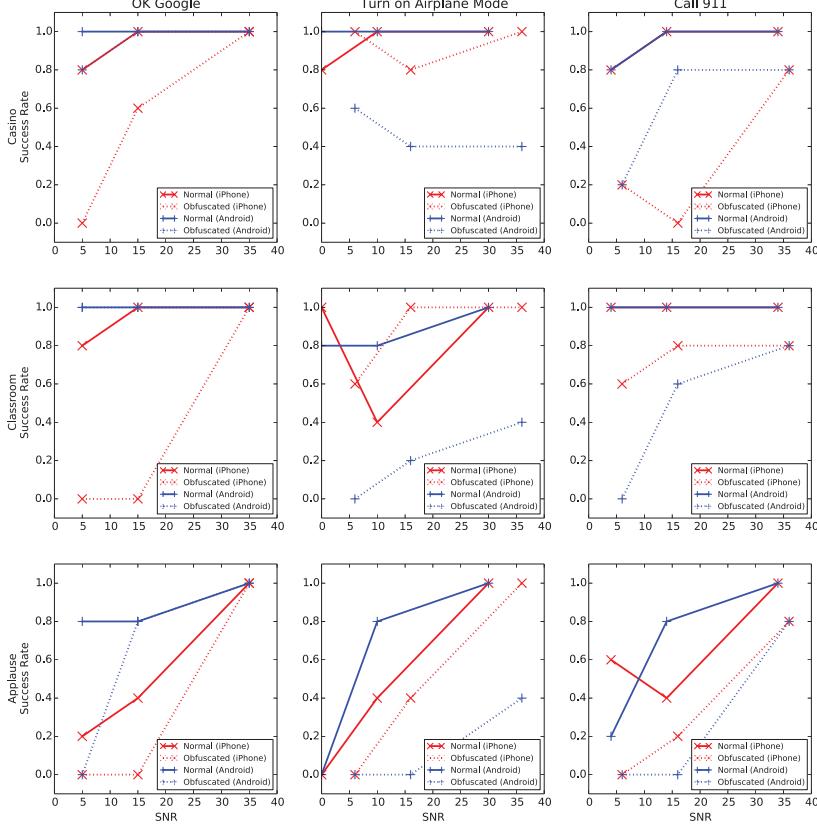


Figure 9: Machine understanding of normal and obfuscated variants of “OK Google”, “Turn on Airplane Mode”, and “Call 911” voice commands (*column-wise*) under different background noises (*row-wise*). Each graph shows the measured average success rate (the fraction of correct transcripts) on the y-axis as a function of the signal-to-noise ratio.

C Analysis of Transcriptions using Phoneme-Based Edit Distance Metrics

C.1 Black-box attack

To verify the results of the white-box survey and to better understand the results of Amazon Mechanical Turk Study, we first performed a simple binary classification of transcription responses provided by Turk workers.

We define *phoneme edit distance* δ as the Levenshtein edit distance between phonemes of two transcriptions. We define ϕ as δ/L , where L is the phoneme length of normal command sentence. The use of ϕ reflects how close the transcriptions might sound to a human listener. $\phi < 0.5$ indicates that the human listener successfully comprehended at least 50% of the underlying voice command. We consider this as successful comprehension by human, implying attack failure; otherwise, we consider it a success for the attacker. Table 6 shows the results of our binary classification. The difference in success rates of normal and obfuscated commands is similar to that of human listeners in Table 2, validating the survey results.

We used *relative phoneme edit distance* to show the gap between transcriptions of normal and obfuscated commands submitted by turk workers. The *relative phoneme edit distance* is calculated as $\delta/(\delta + L)$, L is again the phoneme length of normal command sentence. The relative phoneme edit distance has a range of $[0, 1]$, where 0 indicates exact match and larger relative phoneme edit distances mean the evaluator’s transcription further deviates from the ground truth. By this definition, a value of 0.5 is achievable by transcribing silence. Values above 0.5 indicate no relationship between the transcription and correct audio.

Figure 10 shows the CDF of the relative phoneme edit distance for the (*left*) “OK Google”, (*center*) “Turn on Airplane Mode” and (*right*) “Call 911” voice commands. These graphs show similar results as reported in Table 2: Turk workers were adept at correctly transcribing normal commands even in presence of background noise; over 90% of workers made perfect transcriptions with an edit distance of 0. However, the workers were far less able to correctly comprehend obfuscated commands: less than 30% were able to achieve a relative edit distance less than 0.2 for “OK Google” and “Turn on Airplane Mode”.

Table 6: Black-box attack. Percentages show the fraction of human listeners who were able to comprehend at least 50% of voice commands.

	OK Google	Turn On Airplane Mode	Call 911
Normal	97% (97/100)	89% (102/114)	92% (75/81)
Obfuscated	24% (23/94)	47% (52/111)	95% (62/65)

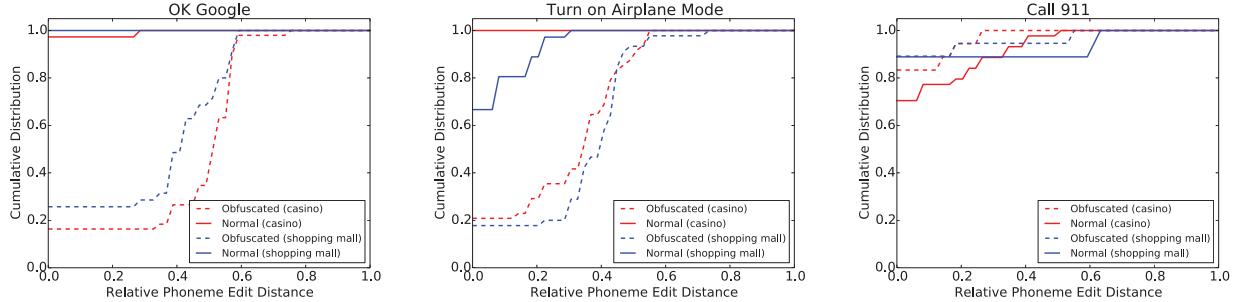


Figure 10: Cumulative distribution of relative phoneme edit distances of Amazon Mechanical Turk workers’ transcriptions for (left) “OK Google”, (center) “Turn on Airplane Mode” and (right) “Call 911” voice commands, with casino and shopping mall background noises. The attack is successful for the first two commands, but fails for the third.

Table 7: White-box attack. Percentages show the fraction of human listeners who were able to comprehend at least 50% of phonemes in a command.

	Command
Normal	97% (297/310)
Obfuscated	10% (37/377)

C.2 White-box attack

To verify the results of our authors review of the Turk study, we computed the edit distance of transcribed commands with actual commands. Table 7 says a command is a match if at least 50% of phonemes were transcribed correctly, to eliminate potential author bias. This metric is less strict both for normal commands and obfuscated commands, but the drop in quality is nearly as strong.

D Canceling out the Beep

Even when constrained to simplistic and conservative mathematical models, it is difficult to cancel out a beep played by a mobile device.

D.1 Two ears difficulties

Setup: The victim has two ears located at points E and F , and a device at point P . The attacker has complete control over a speaker at point A .

Threat model: The attacker has complete knowledge of the setup, including what the beep sounds like, when the beep will begin playing, and the location of all four points E, F, P and A . We assume for simplicity that sound amplitude does not decrease with distance.

The attacker loses the game if the victim hears a sound in either ear. Our question, then, is: *can the attacker cancel out the sound of the beep in both ears simultaneously?* Since sound amplitude does not attenuate with distance, the attacker can focus solely on phase matching: to cancel out a sound, the attacker has to play a signal that is exactly π radians out of phase with the beep. This means the attacker has to know the phase of the signal to a good degree of accuracy.

In our model, canceling out sound at one ear (say E) is easy for the attacker. The attacker knows the distance d_{PE} , and so knows t_{PE} , the time it will take for the sound to propagate from P to E . Similarly, the attacker knows t_{AE} . This is enough to determine the delay that he needs to introduce: he should start playing his signal $\frac{(d_{PE}-d_{AE}) \pmod \lambda}{c}$ (where λ is the wavelength) seconds after the start of the beep (where c is the speed of sound), and the signal he should play from his speaker is the inverse of the beep (an “anti-beep”).

However, people have two ears, and so there will still be some remnant of the beep at the other ear F : the beep will arrive at that ear $\frac{d_{PF}}{c}$ seconds after being played, while the anti-beep will arrive $\frac{d_{AF}}{c}$ seconds after the anti-beep starts, i.e., $\frac{d_{PE}-d_{AE}+d_{AF}}{c}$ seconds after the beep starts. This means that the anti-beep will be delayed by $\frac{d_{PE}-d_{AE}+d_{AF}-d_{PF}}{c}$ seconds compared to the beep.

Therefore, the attacker must be sure that they are placed exactly correctly so that the cancellation occurs at just the right time for both ears. This is the set of points where $(d_{PE} - d_{AE} + d_{AF} - d_{PF}) = 0$. That is, the attacker can be standing anywhere along half of a hyperbola around the user.

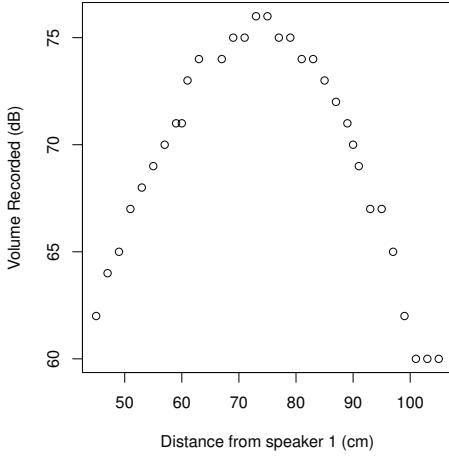


Figure 11: Plot of the amplitude of attempted noise cancellation of a tone at 440Hz

Finally, there is one more issue: any device which can perform voice recognition must have a microphone, and so can therefore listen actively for an attack. This then requires not only that the attacker be able to produce exactly the inverse signal at both ears, but also zero total volume at the device’s location. This then fixes the attacker’s location to only one potential point in space.

D.2 Real-world difficulties

In the above setup we assumed a highly idealized model of the real world. For instance, we assumed that the attacker knows all distances involved very precisely. This is of course difficult to achieve in practice (especially if the victim moves his head). Our calculations show that canceling over 90% of the beep requires an error of at most 3% in the phase. Putting this into perspective, for a 1Khz beep, to eliminate 90% of the noise, the adversary needs to be accurate to within 3 inches.

In practice, the attack is even more difficult than described above. The adversary may have to contend with multiple observers, and has to consider background noise, amplitude attenuation with distance, and so on.

Even so, to investigate the ability of an attacker to cancel sound in near-ideal conditions, we conducted an experiment to show how sound amplitude varies as a function of the phase difference in ideal conditions. The setup is as follows: two speakers are placed facing each other, separated by a distance d . Both speakers play the same pure tone at the same amplitude. We placed a microphone in between, and measured the sound amplitude at various points on the line segment joining the two. For our experiment, $d = 1.5\text{m}$ and the frequency of the tone is $f = 440\text{Hz}$. The results are plotted in Figure 11.

As can be seen, the total cancellation does follow a

sine wave as would be expected, however there is noise due to real-world difficulties. This only makes the attacker’s job more difficult.

E Machine Interpretation of Obfuscated Command

Table 8: For each of the three phrases generated in our white-box attack, the phrase that Sphinx recognized. This data is used to alter the lengths of each phoneme to reach words more accurately. Some words such as “for” and “four” are pronounced exactly the same: Sphinx has no language model and so makes errors here.

Phrases as recognized by CMU Sphinx	
Count	Phrase
3	okay google browse evil dot com
1	okay google browse evil that come
1	okay google browse evil them com
1	okay google browse for evil dot com
6	okay google browse two evil dot com
2	okay google browse two evil that com
1	okay google browse who evil not com
1	okay google browse who evil that com
1	okay up browse evil dot com
5	okay google picture
2	okay google take a picture
1	okay google take of
1	okay google take of picture
6	okay google take picture
10	okay google text one three for five
1	okay google text one two three for five
2	okay google text one who three for five
3	okay google text want three for five

F Short-Term Features used by Classifier Defense

Table 9: Short term features used for extracting mid-term features.

Feature	Description
Zero Crossing Rate	The rate of sign-changes of the signal during the duration of a particular frame.
Energy	The sum of squares of the signal values, normalized by the respective frame length.
Entropy of Energy	The entropy of sub-frames’ normalized energies.
Spectral Centroid	The center of gravity of the spectrum.
Spectral Spread	The second central moment of the spectrum.
Spectral Entropy	Entropy of the normalized spectral energies for a set of sub-frames.
Spectral Flux	The squared difference between the normalized magnitudes of the spectra of the two successive frames.
Spectral Rolloff	The frequency below which 90% of the magnitude distribution of the spectrum is concentrated.
MFCCs	Mel Frequency Cepstral Coefficients
Chroma Vector	A 12-element representation of the spectral energy
Chroma Deviation	The standard deviation of the 12 chroma coefficients.

FlowFence: Practical Data Protection for Emerging IoT Application Frameworks

Earlence Fernandes¹, Justin Paupore¹, Amir Rahmati¹, Daniel Simionato²
Mauro Conti², Atul Prakash¹

¹*University of Michigan* ²*University of Padova*

Abstract

Emerging IoT programming frameworks enable building apps that compute on sensitive data produced by smart homes and wearables. However, these frameworks only support permission-based access control on sensitive data, which is ineffective at controlling *how* apps use data once they gain access. To address this limitation, we present FlowFence, a system that requires consumers of sensitive data to declare their intended data flow patterns, which it enforces with low overhead, while blocking all other undeclared flows. FlowFence achieves this by explicitly embedding data flows and the related control flows within app structure. Developers use FlowFence support to split their apps into two components: (1) A set of Quarantined Modules that operate on sensitive data in sandboxes, and (2) Code that does not operate on sensitive data but orchestrates execution by chaining Quarantined Modules together via taint-tracked *opaque handles*—references to data that can only be dereferenced inside sandboxes. We studied three existing IoT frameworks to derive key functionality goals for FlowFence, and we then ported three existing IoT apps. Securing these apps using FlowFence resulted in an average increase in size from 232 lines to 332 lines of source code. Performance results on ported apps indicate that FlowFence is practical: A face-recognition based door-controller app incurred a 4.9% latency overhead to recognize a face and unlock a door.

1 Introduction

The Internet of Things (IoT) consists of several data-producing devices (*e.g.*, activity trackers, presence detectors, door state sensors), and data-consuming apps that optionally actuate physical devices. Much of this data is privacy sensitive, such as heart rates and home occupancy patterns. More importantly, we are seeing an emergence of application frameworks that enable third party developers to build apps that compute

on such data—Samsung SmartThings [55], Google Brillo/Weave [30], Vera [5], and Apple HomeKit [8] are a few examples.

Consider a smart home app that allows unlocking a door via face recognition using a camera at the door. Home owners may also want to check the state of the door from a secure Internet site (thus, the app requires Internet access). Additionally, the user also wants to ensure that the app does not leak camera data to the Internet. Although this app is useful, it also has the potential to steal camera data. Therefore, enabling apps to compute on sensitive data the IoT generates, while preventing data abuse, is an important problem that we address.

Current approaches to data security in emerging IoT frameworks are modeled after existing smartphone frameworks (§2). In particular, IoT frameworks use permission-based access control for data sources and sinks, but they do not control *flows* between the authorized sources and sinks. This method has already proved to be inadequate, as is evident from the growing reports of data-stealing malware in the smartphone [73] and browser extension spaces [36, 14]. The fundamental problem is that users have no choice but to take it on faith that an app will not abuse its permissions. Instead, we need a solution that forces apps to make their data use patterns explicit, and then enforce the declared information flows, while preventing all other flows.

Techniques like the recognizer OS abstraction [39] could enable privacy-respecting apps by reducing the fidelity of data released to apps so that non-essential but privacy violating data is removed. However, these techniques fundamentally depend on the characteristics of a particular class of applications (§7). For example, image processing apps may not need HD camera streams and, thus, removing detail from those streams to improve privacy is feasible. However, this may not be an option in the general case for apps operating on other types of sensitive data.

Dynamic or static taint analysis has been suggested

as a method to address the limitations of the above permission-based systems [60, 53]. Unfortunately, current dynamic taint analysis techniques have difficulty in dealing with implicit flows and concurrency [59], may require specialized hardware [70, 54, 65], or tend to have significant overhead [48]. Static taint analysis techniques [9, 21, 66, 45] alleviate run-time performance overhead issues, but they still have difficulty in handling implicit flows. Furthermore, some flow-control techniques require developers to use special-purpose languages, for example, JFlow [45].

We present FlowFence, a system that enables robust and efficient flow control between sources and sinks in IoT applications. FlowFence addresses several challenges including not requiring the use of special-purpose languages, avoiding implicit flows, not requiring instruction-level information flow control, supporting flow policy rules for IoT apps, as well as IoT-specific challenges like supporting diverse app flows involving a variety of device data sources.

A key idea behind FlowFence is its new information flow model, that we refer to as *Opacified Computation*. A data-publishing app (or sensitive source) tags its data with a taint label. Developers write data-consuming apps so that sensitive data is only processed within designated functions that run in FlowFence-provided sandboxes for which taints are automatically tracked. Therefore, an app consists of a set of designated functions that compute on sensitive data, and code that does not compute on sensitive data. FlowFence only makes sensitive data available to apps via functions that they submit for execution in FlowFence-provided sandboxes.

When such a function completes execution, FlowFence converts the function’s return data into an *opaque handle* before returning control to the non-sensitive code of the app. An opaque handle has a hidden reference to raw sensitive data, is associated with a taint set that represents the taint labels corresponding to sensitive data accessed in generating the handle, and can only be dereferenced within a sandbox. Outside a sandbox, the opaque handle does not reveal any information about the data type, size, taint label, any uncaught exception in the function, or contents. When a opaque handle is passed as a parameter into another function to be executed in a sandbox, the opaque handle is dereferenced before executing the function, and its taint set added to that sandbox. When a function wants to declassify data to a sink, it makes use of FlowFence-provided Trusted APIs that check $\langle \text{source}, \text{sink} \rangle$ flow policies before declassifying data. The functions operating on sensitive data can communicate with other functions, and developers can chain functions together to achieve useful computations but only through well-defined FlowFence-controlled channels and only through the use of opaque handles.

Therefore, at a high level, FlowFence creates a data flow graph at runtime, whose nodes are functions, and whose edges are either raw data inputs or data flows formed by passing opaque handles between functions. Since FlowFence explicitly controls the channels to share handles as well as declassification of handles (via Trusted API), it is in a position to act as a secure and powerful reference monitor on data flows. Since the handles are opaque, untrusted code cannot predicate on the handles outside a sandbox to create implicit flows. Apps can predicate on handles within a sandbox, but the return value of a function will always be tainted with the taint labels of any data consumed, preventing apps from stripping taint. An app can access multiple sources and sinks, and it can support multiple flows among them, subject to a stated flow policy.

Since sensitive data is accessible only to functions executing within sandboxes, developers must identify such functions to FlowFence—they encapsulate functions operating on sensitive data in Java classes and then register those classes with FlowFence infrastructure. Furthermore, FlowFence treats a function in a sandbox as a blackbox, scrutinizing only communications into and out of it, making taint-tracking efficient.

FlowFence builds on concepts from systems for enforcing flow policies at the component level, for example, COWL for JavaScript [63] and Hails for web frameworks [28, 52]. FlowFence is specifically tailored for supporting IoT application development. Specifically, motivated by our study of three existing IoT application frameworks, FlowFence includes a flexible Key-Value store and event mechanism that supports common IoT app programming paradigms. It also supports the notion of a discretionary flow policy for consumer apps that enables apps to declare their flow policies in their manifest (and thus the policy is visible prior to an app’s deployment). FlowFence ensures that the IoT app is restricted to its stated flow policy.

Our work focuses on tailoring FlowFence to IoT domains because they are still emerging, giving us the opportunity to build a flow control primitive directly into application structure. Flow-based protections could, in principle, be applied to other domains, but challenges are often domain-specific. This work solves IoT-specific challenges. We discuss the applicability of Opacified Computation to other domains in §6.

Our Contributions:

- We conduct a study of three major existing IoT frameworks that span the domains of smart homes, and wearables (i.e. Samsung SmartThings, Google Fit, and Android Sensor API) to analyze IoT-specific challenges and security design issues, and to inform the functionality goals for an IoT application framework (§2).

- Based on our findings we design the Opacified Computation model, which enables robust and efficient source to sink flow control (§3).
- We realize the Opacified Computation model through the design of FlowFence for IoT platforms. Our prototype runs on a Nexus 4 with Android that acts as our “IoT Hub” (§4). FlowFence only requires process isolation and IPC services from the underlying OS, thus minimizing the requirements placed on the hardware/OS.
- We perform a thorough evaluation of FlowFence framework (§5). We find that each sandbox requires 2.7MB of memory on average. Average latency for calls to functions across a sandbox boundary in our tests was 92ms or less. To understand the impact of these overheads on end-to-end performance, we ported three existing IoT apps to FlowFence (§5.2). Adapting these apps to use FlowFence resulted in average size of apps going up from 232 lines to 332 lines of source code. A single developer with no prior knowledge of the FlowFence API took five days total to port all these apps. Macro-benchmarks on these apps (latency and throughput) indicate that FlowFence performance overhead is acceptable: we found a 4.9% increase in latency for an app that performs face recognition, and we found a negligible reduction in throughput for a wearable heart beat calculator app. In terms of security, we found that the flow policies correctly enforce flow control over these three apps (§5.2). Based on this evaluation, we find FlowFence to be a practical, secure, and efficient framework for IoT applications.

2 IoT Framework Study: Platforms and Threats

We performed an analysis of existing IoT application programming frameworks, apps, and their security models to inform FlowFence design, distill key functionality requirements, and discover security design shortcomings. Our study involved analyzing three popular programming frameworks covering three classes of IoT apps: (1) Samsung SmartThings for the smart home, (2) Google Fit for wearables, and (3) Android Sensor API for quantified-self apps.¹ We manually inspected API documentation, and mapped it to design patterns. We found that across the three frameworks, access to IoT sensor data falls in one of the following design patterns: (1) The *polling pattern* involving apps polling an IoT device’s current state; and (2) The *callback pattern* involving

¹Quantified Self refers to data acquisition and processing on aspects of a person’s daily life, e.g., calories consumed.

apps registering callback functions that are invoked whenever an IoT device’s state changes.²

We also found that it is desirable for publishers and consumers to operate in a device-agnostic way, without being explicitly connected to each other, *e.g.*, a heart rate monitor may go offline when a wearable is out of Bluetooth range; the consumer should not have to listen to lifecycle events of the heart rate monitor—it only needs the heart beat data whenever that is available. Ideally, the consumer should only need to specify the type of data it requires, and the IoT framework should provide this data, while abstracting away the details. Furthermore, this is desirable because there are many types of individual devices that ultimately provide the same kind of data, *e.g.*, there are many kinds of heart rate monitors eventually providing heart rate data.

A practical IoT programming framework should support the two data sharing patterns described above in a device-agnostic manner. In terms of security, we found that all three frameworks offer permission-based access control, but they do not provide any methods to control data use once apps gain access to a resource. We provide brief detail on each of these frameworks below.

1) Samsung SmartThings. SmartThings is a smart home app programming framework [4] with support for 132 device types ranging from wall plugs to ZWave door locks. SmartThings provides two types of APIs to access device data: `subscribe` and `poll`. The `subscribe` API is the callback design pattern. For instance, to obtain a ZWave door lock’s current state, an app would issue a call of the form `subscribe(lockDevice, "lock.state", callback)`. The `subscribe` API abstracts away details of retrieving data from a device, and directly presents the data to consumers, allowing them to operate in a disconnected manner. The `poll` API is the polling pattern. For example, an app can invoke `lockDevice.currentState` to retrieve the state of the lock at that point in time.

For permission control, the end-user is prompted to authorize an app’s access request to a device [57], based on a matching of *SmartThings capabilities* (a set of operations) that the app wishes to perform, and the set of capabilities that a device supports. Once an app is granted access to a device, it can access all of its data and features. SmartThings does not offer any data flow control primitives.

2) Google Fit. Google Fit enables apps to interface with wearables like smartwatches [32]. The core abstraction in Google Fit is the Fitness Data Type, which provides a

²We also found an orthogonal *virtual sensor* design pattern: An intermediate app computing on sensor data and re-publishing the derived data as a separate virtual sensor. For instance, an app reads in heart rate at beats-per-minute, derives beats-per-hour, and re-publishes this data as a separate sensor.

device-agnostic abstraction for apps to access them in either instantaneous or aggregated form. The API provides raw access to both data types using only the callback pattern; the polling pattern is not supported. For instance, to obtain expended calories, an app registers a data point listener for the `com.google.calories.expended` instantaneous fitness type. A noteworthy aspect is that apps using the Fit API can pre-process data and publish *secondary* data sources, essentially providing a virtual sensor.

Google Fit API defines *scopes* that govern access to fitness data. For instance, the `FITNESS_BODY_READ` scope controls access to heart rate. Apps must request read or write access to a particular scope, and the user must approve or deny the request. Once an app gains access to a scope, it can access all fitness related data in that scope. Google Fit does not offer any data flow control primitives.

3) Android Sensor API. Android provides API access to three categories of smartphone sensor data: Motion, Environment, and Position. Apps must register a class implementing the `SensorEventListener` interface to receive callbacks that provide realtime sensor state. There is no API to poll sensor state, except for the Location API. Android treats the Location API differently but, for our purposes, we consider it to be within the general umbrella of the sensor API. The Location API supports both the polling and callback design patterns. The callback pattern supports consumers operating in a device-agnostic manner since the consumer only specifies the type of data it is interested in.

Surprisingly, the Android sensor API does not provide any access control mechanism protecting sensor data. Any app can register a callback and receive sensor data. The Location API and heart rate sensor API, however, do use Android permissions [22, 31]. Similar to the previous two frameworks, Android does not offer any data flow control primitives.

IoT Architectures. We observe two categories of IoT software architectures: (1) Hub, and (2) Cloud. The hub model is centralized and executes the majority of software on a hub that exists in proximity to various physical devices, which connect to it. The hub has significantly more computational power than individual IoT devices, has access to a power supply, provides network connectivity to physical devices, and executes IoT apps. In contrast, a cloud architecture executes apps in remote servers and may use a minimal hub that only serves as a proxy for relaying commands to physical devices. The hub model is less prone to reliability issues, such as functionality degradation due to network connectivity losses that plague cloud architectures [58]. Furthermore, we observe a general trend toward adoption of the hub model by industry in systems such as Android Auto [1]

and Wear [2], Samsung SmartThings [55]³, and Logitech Harmony [3]. Our work targets the popular hub model, making it widely applicable to these hub-based IoT systems.

Threat Model. IoT apps are exposed to a slew of sensitive data from sensors, devices connected to the hub, and other hub-based apps. This opens up the possibility of sensitive data leaks leading to privacy invasion. For instance, Denning *et al.* outlined emergent threats to smart homes, including misuse of sensitive data for extortion and for blackmail [17]. Fernandes *et al.* recently demonstrated that such threats exist in real apps on an existing IoT platform [26] where they were able to steal and misuse door lock pincodes.

We assume that the adversary controls IoT apps running on a hub whose platform software is trusted. The adversary can program the apps to attempt to leak sensitive data. Our security goal is to force apps to declare their intended data use patterns, and then enforce those flows, while preventing all other flows. This enables the design of more privacy-respecting apps. For instance, if an app on FlowFence declares it will sink camera data to a door lock, then the system will ensure that the app cannot leak that data to the Internet. We assume that side channels and covert channels are outside the scope of this work. We discuss implications of side channels, and possible defense strategies in §6.

3 Opacified Computation Model

Consider the example smart home app from §1, where it unlocks the front door based on people’s faces. It uses the bitmap to extract features, checks the current state of the door, unlocks the door, and sends a notification to the home owner using the Internet. This app uses sensitive camera data, and accesses the Internet for the notification (in addition to ads and crash reporting). An end user wishes to reap the benefits of such a scenario but also wants to ensure that the door control app does not leak camera data to the Internet.

FlowFence supports such scenarios through the use of Opacified Computation, which consists of two main components: (1) Quarantined Modules (“functions”), and (2) opaque handles. A Quarantined Module (QM) is a developer-written code module that computes on sensitive data (which is assigned a taint label at the data source), and runs in a system-provided sandbox. A developer is free to write many such Quarantined Modules. Therefore, each app on FlowFence is split into two parts: (1) some non-sensitive code that does not compute on sensitive data, and (2) a set of QMs that compute on sensitive data. Developers can chain multiple QMs together

³Recent v2 hubs have local processing.

to achieve useful work, with the unit of transfer between QMs being opaque handles—immutable, labeled opaque references to data that can only be dereferenced by QMs when running inside a sandbox. QMs and opaque handles are associated with a taint set, *i.e.*, a set of taint labels that indicates the provenance of data and helps track information flows (we explain label design later in this section).

An opaque handle does not reveal any information about the data value, data type, data size, taint set, or exceptions that may have occurred to non-sensitive code. Although such opaqueness can make debugging potentially difficult, our implementation does support a development-time debugging flag that lifts these opaqueness restrictions (§4).

Listings 1 and 2 shows pseudo-code of example smart home apps. The CamPub app defines `QM_bmp` that publishes the bitmap data. FlowFence ensures that whenever a QM returns to the caller, its results are converted to an opaque handle.

Line 10 of Listing 1 shows the publisher app calling the QM (a blocking call), supplying the function name and a taint label. FlowFence allocates a clean sandbox, and runs the QM. The result of `QM_bmp` running is the opaque handle `hCam`, which refers to the return data, and is associated with the taint label `Taint_CAMERA`. `hCam` is immutable—it will always refer to the data that was used while creating it (immutability helps us reduce overtainting; we discuss it later in this section). Line 11 shows CamPub sending the resultant handle to a consumer.

We also have a second publisher of data `QM_status` that publishes the door state (Line 16 of Listing 1), along with a door identifier, and provides an IPC function for consumers to call (Line 20).

The DoorCon app defines `QM_recog`, which expects a bitmap, and door state (Lines 6-9 of Listing 2). It computes feature vectors from the bitmap, checks if the face is authorized, checks the door state, and unlocks the door. Lines 18, 19 of Listing 2 show this consumer app receiving opaque handles from the publishers. As discussed, non-sensitive code only sees opaque handles. In this case, `hCam` refers to camera-tainted data, and `hStatus` refers to door-state-tainted data, but the consumer app cannot read the data unless it passes the data to a QM. Moreover, for this same reason, non-sensitive code cannot test the value of a handle to create an implicit flow.

Line 20 calls a QM, passing the handles as parameters. FlowFence automatically and transparently dereferences opaque handle arguments into raw data before invoking a QM. Transparent dereferencing of opaque handles offers developers the ability to write QMs normally with standard types even though some parameters may be passed as opaque handles. During this process, FlowFence allocates a clean sandbox for the QM to run, and propagates

the taint labels of the opaque handles to that sandbox. Finally, `QM_recog` receives the raw data and opens the door.

The consumer app uses `QM_report` to send out the state of the door to a remote monitoring website. It also attempts to use `QM_mal` to leak the bitmap data. FlowFence prevents such a leak by enforcing flow policies, which we discuss next.

Flow Policy. A publisher app, which is associated with a sensor (or sensors), can add taint labels to its data that are tuples of the form $(appID, name)$, where $appID$ is the identifier of the publisher app and $name$ is the name of the taint label. This name denotes a standardized type that publishers and consumers can agree upon, for example, `Taint_CAMERA`. We require labels to be statically declared in the app’s manifest. $appID$ is unique to an app and is used to avoid name collisions across apps.⁴ Additionally, in its manifest, the publisher can specify a set of flow rules for each of its taint labels, with the set of flow rules constituting the publisher policy. The publisher policy defines the permissible flows that govern the publisher’s data. A flow rule is of the form `TaintLabel → Sink`, where a sink can be a user interface, actuators, Internet, etc. CamPub’s flow policy is described on Line 3 of Listing 1. The policy states that consumer apps can sink camera data to the sink labeled `UI` (which is a standard label corresponding to a user’s display at the hub).

Since other possible sinks for camera data are not necessarily known to the publisher, new flow policies are added as follows. A consumer app must request approval for flow policies if it wants to access sensitive data. Consumer flow policies can be more restrictive than publisher policies, supporting the least privilege principle. They can also request new flows, in which case the hub user must approve them. DoorCon’s policy requests are described in Lines 2-4 of Listing 2. It requests the flows: `Taint_CAMERA → Door.Open`, `Taint_DOORSTATE → Door.Open`, `Taint_DOORSTATE → Internet`. At app install time, a consumer app will be bound to a publisher that provides data sources with labels `Taint_CAMERA`, `Taint_DOORSTATE`.

To compute the final policy for a given consumer app FlowFence performs two steps. First, it computes the intersection between the publisher policy and the consumer policy flow rules. In our example, the intersection is the null set. If it were not null, FlowFence would authorize the intersecting flows for the consumer app in question. Second, it computes the set difference between the consumer policy and publisher policy. This difference reflects the flows the consumer has requested but the publisher policy has not covered. At this point, FlowFence

⁴ An app cannot forge its ID since our implementation uses Android package name as the ID. See §4 for details.

delegates approval to the IoT hub owner to make the final decision about whether to approve the flows or not. If the hub owner decides to approve a flow that a publisher policy does not cover, that exception is added for subsequent runs of that consumer app. Such a approval does not apply to other apps that may also use the data.

If a QM were to attempt to declassify the camera data to the Internet (*e.g.*, `QM_mal`) directly without requesting a flow policy, the attempt would be denied as none of the flow policies allow it. An exception is thrown to the calling QM whenever it tries to perform an unauthorized declassification. Similar to exception processing in languages like Java, if a QM does not catch an exception, any output handle of this QM is moved into the exception state. Non-QM code cannot view this exception state. If an app uses such a handle in a subsequent QM as a parameter, then that QM will silently fail, with all of its output handles also in the exception state. App developers can avoid this by ensuring that a QM handles all possible exceptions before returning and, if necessary, encodes any errors into the return object, which can then be examined in a subsequent QM that receives the returned handle.

FlowFence is in a position to make security decisions because the publisher assigns taint labels while creating the handles, and when `DoorCon` reads in the handles, it results in the taint labels propagating to the sandbox running `QM_mal`. FlowFence simply reads the taint labels of the sandbox at the time of declassification.

All declassification of sensitive data can only occur through well-known trusted APIs that FlowFence defines. Although our prototype provides a fixed set of trusted APIs that execute in a separate trusted process, we envision a plug-in architecture that supports community built and vetted APIs (§4). FlowFence sets up sandbox isolation such that attempts at declassifying data using non-trusted APIs, such as arbitrary OS system calls, are denied.

Table 1 summarizes the taint logic. When a clean sandbox loads a QM, it has no taint. A taint label, belonging to the app, may be added to a handle at creation, or to a sandbox at any time, allowing data providers to label themselves as needed. A call from QM executing in S_0 to another QM that is launched in sandbox S_1 results in the taint labels of S_0 being copied to S_1 . When a called QM returns, FlowFence copies the taint of the sandbox into the automatically created opaque handle. At that point, the QM no longer exists. The caller is not tainted by the returned handle, unless the caller (which must be a QM) dereferences the handle. These taint arithmetic rules, combined with QMs, opaque handles, and sandboxes conceptually correspond to a directed data flow graph from sources to sinks, as we illustrate with the example below.

```

1 application CamPub
2 taint_label Taint_CAMERA;
3 allow { Taint_CAMERA -> UI }
4
5 Bitmap QM_bmp():
6     Bitmap face = camDevice.snapshot();
7     return face;
8
9 if (motion at FrontDoor)
10    hCam = QM.call(QM_bmp, Taint_CAMERA);
11    send hCam to DoorCon;
12 -----
13 application DoorStatePub
14 taint_label Taint_DOORSTATE;
15
16 Status QM_status():
17     return (door[0].state(), 0); //state, idx
18
19 /* IPC */ Handle getDoorState():
20     return QM.call(QM_status,
21                     Taint_DOORSTATE);

```

Listing 1: Pseudocode for two publishers—camera data, and door state. Quarantined Modules are shown in light gray.

```

1 application DoorCon
2 request { Taint_CAMERA -> Door.Open,
3             Taint_DOORSTATE -> Door.Open,
4             Taint_DOORSTATE -> Internet }
5
6 void QM_recog(faceBmp, status):
7     Features f = extractFeatures(faceBmp);
8     if(status != unlocked AND isAuthenticated(f))
9         TrustedAPI.door[0].open();
10
11 void QM_report(status):
12     TrustedAPI.network.send(status);
13
14 void QM_mal(faceBmp):
15     /* this is denied */
16     TrustedAPI.network.send(faceBmp);
17
18 receive hCam from CamPub;
19 Handle hStatus =
20     DoorStatePub.getDoorState();
21 QM.call(QM_recog, hCam, hStatus);
22 QM.call(QM_mal, hCam);
23 QM.call(QM_report, hStatus);

```

Listing 2: Consumer app pseudocode that reads camera and door state data, and controls a door. Quarantined Modules are shown in light gray.

FlowFence Data Flow Graph. We now discuss the taint flow logic of FlowFence in more detail, and show how it creates and tracks, at runtime, a directed data flow graph that enables it to make security decisions on flows. Figure 1 shows two publishers of sensitive data that generate $OH_{T_1}(d_1)$ —an opaque handle that refers to camera bitmap data d_1 , and $OH_{T_2}(d_2)$ —an opaque handle that refers to door state data d_2 , using QM_{bmp} and QM_{status}

Operation	Taint Action
Sandbox S loads a QM	$T[S] := \emptyset$
QM inside S reads opaque handle $d = OH^{-1}(h)$	$T[S] += T[h]$
QM inside S returns $h = OH(d)$	$T[h] := T[S]$
QM manually adds taints $\{t\}$ to its sandbox	$T[S] += \{t\}$
QM_0 inside S_0 calls QM_1 inside S_1	$T[S_1] = T[S_0]$

Table 1: Taint Arithmetic in FlowFence. $T[S]$ denotes taint labels of a sandbox running a QM. $T[h]$ denotes taint label of a handle h .

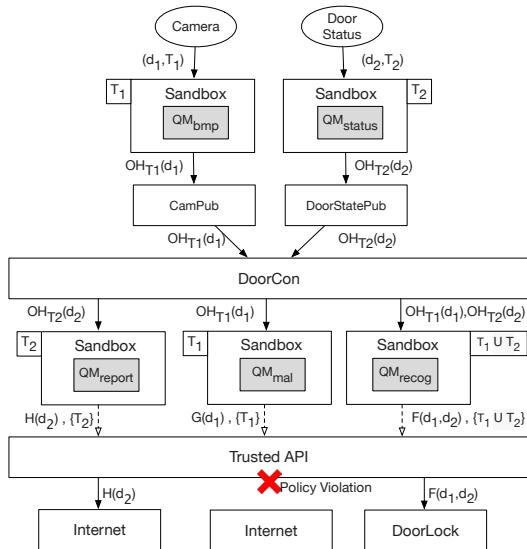


Figure 1: Data flow graph for our face recognition example. FlowFence tracks taint labels as they propagate from sources, to handles, to QMs, to sinks. The dotted lines represent a declassification attempt. The trusted API uses labels on the sandboxes to match a flow policy.

respectively. T_1 and T_2 are taint labels for data d_1 and d_2 . The user wants to ensure that camera data does not flow to the internet.

The consumer app (DoorCon) consists of non-sensitive code that reads the above opaque handles from the publishers, and invokes three QMs. QM_{recog} operates on both $OH_{T_1}(d_1)$ and $OH_{T_2}(d_2)$. When the non-sensitive code requests execution of QM_{recog} , FlowFence will allocate a clean sandbox, dereference the handles into raw values, and invoke the module. The sandbox inherits the taint label $T_1 \cup T_2$. Later on, when QM_{recog} tries to declassify its results by invoking the trusted API, FlowFence will read the taint labels (dotted line in Figure 1)— $T_1 \cup T_2$. That is, FlowFence taint arithmetic defines that the taint label of the result is the combination of input

data taint labels. In our example, declassifying camera and door state tainted data to the door lock is permitted, since the user authorized the flow earlier.

If the consumer app tries to declassify sensitive data d_1 by invoking a trusted API using QM_{mal} , the API reads the taint labels on the handle being declassified, determines that there is no policy that allows $d_1 \rightarrow \text{Internet}$, and denies the declassification.

Immutable opaque handles are key to realizing this directed data flow graph. Consider Figure 1. If handles were mutable, and if QM_{mal} read in some data with taint label T_3 , then we would have to assume that $OH_{T_1}(d_1)$ is tainted with T_3 , leading to overtainting. Later on, when QM_{recog} executes, its sandbox would inherit the taint label T_3 due to the overtainting. If there was a policy that prevented T_3 from flowing to the door lock, FlowFence would prevent QM_{recog} from executing the declassification. FlowFence avoids these overtainting issues by having immutable handles, which enable better precision when reasoning about flows. There are other sources of overtainting related to how a programmer structures the computation and IoT-specific mechanisms that FlowFence introduces. We discuss their implications and how to manage them in §4 and §6.

As discussed above, taint flows transitively from data sources, to opaque handles, to sandboxes, back to opaque handles, and eventually to sinks via the trusted API, where FlowFence can enforce security policies. This design allows taint flow to be observed in a black-box manner, simply by tracking the inputs and outputs. This allows QMs to internally use any language, without the overhead of native taint tracking, only by using sandbox processes to enforce isolation as described in §4.

FlowFence Security Guarantees. FlowFence uses its taint arithmetic rules to maintain the invariant that the taint set of a QM executing in a sandbox at any time represents the union of the taints of sensitive data used by the QM through opaque handles or through calls from another QM. Furthermore, FlowFence avoids propagating taint on QM returns with the help of opaque handles. Since these handles are opaque outside a QM, non-sensitive code must pass them into QMs to dereference them, allowing FlowFence to track taints. If the non-sensitive code of a consumer app transmits an opaque handle to another app via an OS-provided IPC mechanism, FlowFence still tracks that flow since the receiving app also has to use a QM to make use of the handle.

To prevent flow policy violations, a sandbox must be designed such that writes from a QM to a sink go through a trusted API that enforces specified flow policies. We discuss how we achieve this sandbox design in §4.

4 FlowFence Architecture

FlowFence supports executing untrusted IoT apps using two major components (Figure 2): (1) A series of *sandboxes* that execute untrusted, app-provided QMs in an isolated environment that prevents unwanted communication, and (2) A *Trusted Service* that maintains handles and the data they represent; converting data to opaque handles and dereferencing opaque handles back; mediating data flow between sources, QMs, and sinks, including taint propagation and policy enforcement; and creating, destroying, scheduling, and managing lifetime of sandboxes.

We discuss the design of these components in the context of an IoT hub with Android OS running on top. We selected Android because of the availability of source code. Google’s recently announced IoT-specific OS—Brillo [29], is also an Android variant.⁵ Furthermore, with the introduction of Google Weave [30], we expect to see Android apps adding IoT capabilities in the future.

Untrusted IoT Apps & QMs. Developers write apps for FlowFence in Java and can optionally load native code into QMs. As shown in Figure 2, each app consists of code that does not use sensitive data inputs, and a set of QMs that use sensitive data inputs. Although abstractly, QMs are functions, we designed them as methods operating on serializable objects. Each method takes some number of parameters, each of which can either be (1) raw, serialized data, or (2) opaque handles returned from previous method calls on this or another QM. A developer can write a method to return raw data, but returning raw data would allow leakage. Thus, FlowFence converts that raw data to an opaque handle prior to returning to the untrusted app.⁶

Trusted Service & APIs. This service manages all sensitive data flowing to and from QMs that are executing in sandboxes. It schedules QMs for execution inside sandboxes, dereferencing any opaque handle parameters, and assigning the appropriate taint labels to the sandboxes. The Trusted Service also ensures that a sandbox is correctly tainted whenever a QM reads in sensitive data (Tainter component of Figure 2), as per the taint arithmetic rules in FlowFence (Table 1). Once it taints a sandbox, the Trusted Service maintains the current taint labels securely in its process memory.

FlowFence does not track or update taints for variables inside a QM. Instead, it treats a QM as a blackbox for the

⁵Brillo OS is only a limited release at the time of writing. Therefore, we selected the more mature codebase for design, since core services are the same on Android and Brillo.

⁶A QM can theoretically leak sensitive data through side channels (*e.g.*, by varying the execution time of the method prior to returning). We assume side channels to be out of scope of our system and thus we do not address them in our current threat model. If such leaks were to be a concern, we discuss potential defense strategies in §6.

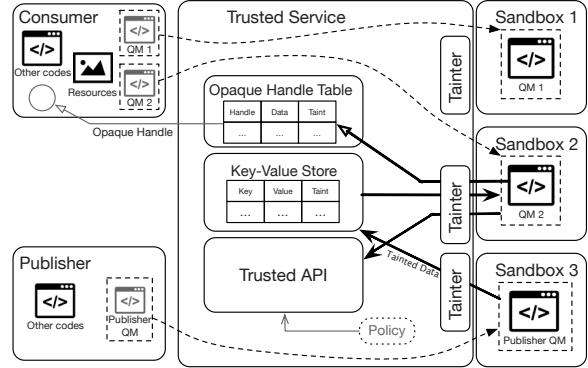


Figure 2: FlowFence Architecture. Developers split apps into Quarantined Modules, that run in sandbox processes. Data leaving a sandbox is converted to an opaque handle tainted with the sandbox taint set.

purpose of taint analysis and it only needs to examine sensitive inputs being accessed or handles provided to a method as inputs. We expect QMs to be limited to the subset of code that actually processes sensitive data, with non-sensitive code running without change. Although this does reduce performance overhead and avoids implicit flow leaks by forcing apps to only use controlled and well-defined data transfer mechanisms, it does require programmers to properly split their app into least-privilege QMs, which if done incorrectly, could lead to overtainting.

When a QM Q running inside a sandbox S returns, the Trusted Service creates a new opaque handle h corresponding to the return data d , and then creates an entry $\langle h, \langle d, T[S] \rangle \rangle$ in its opaque handle Table (Figure 2), and returns h to the caller.

The Trusted Service provides APIs for QMs allowing them to access various sinks. Our current prototype has well-known APIs for access to network, ZWave switches, ZWave locks, camera streams, camera pictures, and location. As an example of bridging FlowFence with such cyber-physical devices, we built an API for Samsung SmartThings. This API makes remote calls to a web services SmartThings app that proxies device commands from FlowFence to devices like ZWave locks. The Trusted API also serves as a policy enforcement point, and makes decisions whether to allow or deny flows based on the specific policy set for the consumer app.

We envision a plug-in architecture that enables community-built and vetted Trusted APIs to integrate with our framework. The plugin API should ideally be in a separate address space. The Trusted Service will send already declassified data to this plugin API via secure IPC. This limits risk by separating the handle table from external code.

Sandboxes. The Trusted Service uses operating system support to create sandbox processes that FlowFence uses to execute QMs. When a QM arrives for execution, FlowFence reserves a sandbox for exclusive use by that QM, until execution completes. Once a QM finishes executing, FlowFence sanitizes sandboxes to prevent data leaks. It does this by destroying and recreating the process.

For efficiency reasons, the Trusted Service maintains a pool of clean spare sandboxes, and will sanitize idle sandboxes in the background to keep that pool full. In addition, the Trusted Service can reassign sandboxes without needing to sanitize them, if the starting taint (based on the input parameters) of the new QM is a superset of or equal to the ending taint of the previous occupant of that sandbox. This is true in many common cases, including passing the return value of one QM directly into another QM. In practice, sandbox restarts only happen on a small minority of calls.

FlowFence creates the sandboxes with the `isolatedProcess` flag set, which causes Android to activate a combination of restrictive user IDs, IPC limitations, and strict SELinux policies. These restrictions have the net effect of preventing the isolated process from communicating with the outside world, except via an IPC interface connected to the Trusted Service.

As shown in Figure 2, this IPC interface belongs to the Trusted API discussed earlier. When the sandboxes communicate with the Trusted Service over an IPC interface, the IPC request is matched to the sandbox it originated from as well as to the QM that initiated the call. As discussed, the Trusted Service maintains information about each sandbox, including its taint labels and running QM, in a lookup table in its own memory, safely out of reach of, possibly malicious, QMs.

Debugging. Code outside QMs cannot dereference opaque handles to inspect corresponding data or exceptions, complicating debugging during development. To alleviate this, FlowFence supports a development time debugging option that allows code outside a QM to dereference handles and inspect their data and any exception traces. However, a deployment of FlowFence has this debugging flag removed. Also, as discussed previously, use of a opaque handle in exception state as a parameter to a QM results in the QM returning a new opaque handle that is also in the exception state. Providing a mechanism for exception handling in the called QM without increasing programmer burden is challenging and a work-in-progress. Currently, we use the idiom of a QM handling all exceptions it can and encoding any error as part of the returned value. This allows any subsequent QM that is called with the handle as a parameter to examine the value and handle the error.

Key-Value Store. This is one of the primary data-sharing mechanisms in FlowFence between publishers of tainted sensitive data and consumer apps that use the data. This design was inspired by our framework study in §2, and it supports publishers and consumers operating in a device-agnostic manner, with consumers only having to know the type of data (taint label) they are interested in processing. Each app receives its own KV store (Figure 2) into which it can update the value associated with a key by storing a $\langle key, sensitive_value, taint_label \rangle$ while executing a QM. For instance, a camera image publisher may create a key such as `CAM_BITMAP`, with an image byte array as the value, and a taint label `Taint_CAMERA` to denote the type of published data (declared in the app manifest). A key is public information—non-sensitive code outside a QM must create a key before it can write a corresponding value. This ensures that a publisher cannot use creation of keys as a signaling mechanism. An app on FlowFence can only write to its own KV store. Taints propagate as usual when a consumer app keys from the KV store. Finally, the publishing QM associated with a sensor usually would not read other sensitive information sources, and thus would not have any additional taint. In the case this QM has read other sources of information, then the existing taint is applied to any published data automatically.

If a QM reads a key’s value, the value’s taint label will be added to that QM’s sandbox. All key accesses are pass-by-value, and any subsequent change in a value’s taint label does not affect the taint labels of QMs that accessed that value in prior executions. Consider an example value V with taint label T_1 . Assuming a QM Q_1 accessed this value, it would inherit the taint. Later on, if the publisher changes the taint label of V to $T_1 \cup T_2$, this would not affect the taint label of Q_1 , until it reads V again.

The polling design pattern is easy to implement using a Key-Value Store. A consumer app’s QM can periodically access the value of a given key until it finds a new value or a non-null value. Publicly accessible keys simplify making sensitive data available to third-party apps, subject to flow policies.

Event Channels. This is the second data-sharing mechanism in FlowFence; it supports the design pattern of registering callbacks for IoT device state changes (*e.g.*, new data being available). The channel mechanism supports all primitive and serializable data types. An app creates channels statically by declaring them in a manifest file at development time (non-sensitive code outside QMs could also create it), making it the owner for all declared channels. Once an app is installed, its channels are available for use—there are no operations to explicitly open or close channels. Other app’s QMs can then register to such channels for updates. When a channel-owner’s

QM puts data on the channel, FlowFence invokes all registered QMs with that data as a parameter. FlowFence automatically assigns the current set of taint labels of the channel-owner to any data it puts on the channel, so that all QMs that receive the callback will be automatically tainted correctly. If a QM is executed as a callback for a channel update, it does not return any data to the non-sensitive code of the app.

Although the publishers and consumers can share opaque handles using OS-provided sharing mechanisms, we designed the Key-Value store, and Event channels explicitly so that publishers and consumers can operate in a device-agnostic manner by specifying the types of data they are interested in, ignoring lower level details.

As described here, both inter-app communication mechanisms, the KV store and event channels, can potentially lead to poison-pill attacks [37] where a compromised or malicious publisher adds arbitrary taint labels, with the goal of overtainting consumers and preventing them from writing to sinks. See the discussion of overtainting in §6 for a defense strategy.

FlowFence Policies and User Experience. In our prototype, users install the app binary package with associated policies. FlowFence prompts users to approve consumer flow policies that are not covered by publisher policies at install time. This install-time prompting behavior is similar to the existing Android model. FlowFence models its flow request UI after the existing Android runtime permission request screens, in an effort to remain close to existing permission-granting paradigms and to leverage existing user training with permission screens. However, unlike Android, FlowFence users are requested to authorize flows rather than permissions, ensuring their control over how apps use data. If a user approves a set of flows, FlowFence guarantees that only those flows can occur.

Past work has shown that users often do not comprehend or ignore prompts [25], however, existing research does point out interesting directions for future work in improving such systems. Felt *et al.* discuss techniques to better design prompting mechanisms [23], and Roesner *et al.* discuss contextual prompting [50, 51] as possible improvements.

5 Evaluation

We evaluated FlowFence from multiple perspectives. First, we ran a series of microbenchmarks to study call latency, serialization overhead, and memory overhead of FlowFence. We found that FlowFence adds modest computational and memory costs. Running a sandbox takes $2.7MB$ RAM on average, and running multiple such sandboxes will fit easily within current hardware

for IoT hubs.⁷ We observed a $92ms$ QM call latency with 4 spare sandboxes, which is comparable to the latency of common network calls in IoT apps. FlowFence supports a maximum bandwidth of $31.5MB/s$ for transferring data into sandboxes, which is large enough to accommodate typical IoT apps. Second, we ported three IoT apps to FlowFence to examine developer effort, security, and impact of FlowFence on macro-performance factors. Our results show that developers can use FlowFence with modest changes to their apps and with acceptable performance impact, making FlowFence practical for building secure IoT apps. Porting the three apps required adding 99 lines of code on average per app. We observed a 4.9% latency increase to perform face recognition in a door controller app. More details follow.

5.1 Microbenchmarks

We performed our microbenchmarks on an LG Nexus 4 running FlowFence on Android 5.0. The Nexus 4 serves as our “IoT hub” that runs QMs and enforces flow policies. In our experiments, we evaluated three factors that can affect apps running on FlowFence.

Memory overhead. We evaluated memory overhead of FlowFence using the `MemoryInfo` API. We ran FlowFence with 0 – 15 empty sandboxes and recorded the memory consumption. Our results show that the FlowFence core requires $6.35MB$ of memory while each sandbox requires $2.7MB$ of memory on average. To put this in perspective, LG Nexus 4 has 2GB memory and loading a blank page on the Chrome browser on it used $98MB$ of memory, while loading FlowFence with 16 sandboxes used $49.5MB$. Therefore, we argue that the memory overhead of FlowFence is within acceptable limits for the platform.

QM Call Latency. We measured QM call latency for non-tainted and tainted parameters (30 trials each with 100 QM call-reply sequences) to assess performance in scenarios that allowed reuse of a sandbox without sanitizing and those that required sanitizing. For tainted calls, each QM takes a single boolean parameter that is tainted. We also varied the number of clean spare sandboxes that are available for immediate QM scheduling initially before each trial. Regardless of the number of spare sandboxes, untainted calls (which did not taint the sandboxes and thus could reuse them without sanitizing) showed a consistent latency of $2.1ms$ ($SD=0.4ms$). The tainted calls were made so as to always require a previously-tainted sandbox to be sanitized. Figure 3 shows average latency of tainted calls across 30 trials for different number of spare sandboxes. As the number of spare sandboxes increases from 0 to 4, the average call

⁷For example, Samsung SmartThings hub has $512MB$ RAM [56], and Apple TV hub has $1GB$ RAM [7].

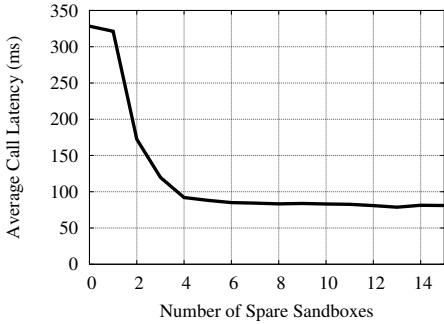


Figure 3: QM Call latency of FlowFence given various number of spare sandboxes, for calls that require previously-used sandboxes to be sanitized before a call. Calls that can reuse sandboxes without sanitizing (untainted calls in our tests) show a consistent latency of 2.1ms , which is not shown in this graph.

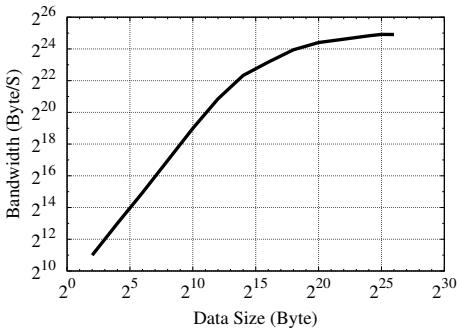


Figure 4: Serialization bandwidth for different data sizes. Bandwidth caps off at 31.5MB/s .

latency decreases from 328ms to 92ms . Further increase in the number of spare sandboxes does not improve latency of QM calls. At 4 spares, the call latency is less than 100ms , making it comparable to latencies seen in controlling many IoT devices (*e.g.*, Nest, SmartThings locks) over a wide-area network. This makes QMs especially suitable to run existing IoT apps that already accept latencies in this range.

Serialization Overhead. To understand FlowFence overhead for non-trivial data types, we computed serialization bandwidth for calls on QMs that cross sandbox boundaries with varying parameter sizes. Figure 4 presents the results for data ranging from 4B to 16MB . The bandwidth increases as data size increases and caps off at 31.5MB/s . This is large enough to support typical IoT apps—for example, the Nest camera uses a maximum bandwidth of 1.2Mbps under high activity [33]. A single camera frame used by one of our ported apps (see below), is 37kB , requiring transferring data at 820kB/s to a QM.

5.2 Ported IoT Applications

We ported three existing IoT apps to FlowFence to measure its impact on security, developer effort, end-to-end latency, and throughput on operations relevant to the apps (Table 2). SmartLights is a common smart home app (*e.g.*, available in SmartThings) that computes a predicate based on a location value from a beacon such as a smartphone, or car [47]. If the location value inside the home’s geofence, the app turns on lights (and adjusts other devices like thermostats) around the home. When the location value is outside the home’s geofence, the app takes the reverse action.

FaceDoor performs face recognition and unlocks a door, if a detected face is authorized [34]. The app uses the camera to take an image of a person at the door, and runs the Qualcomm face recognition SDK (chipset-specific native code, available only as a binary).

HeartRateMonitor accesses a camera to compute heart rate using photoplethysmography [67]. The app uses image processing code on streamed camera frames.

FlowFence provides trusted API to access switches, locks, and camera frames. These three existing apps cover the popular IoT areas of smart homes and quantified self. Furthermore, face recognition and camera-frame-streaming apps are among the more computationally expensive types of IoT apps, and stress test FlowFence performance. We ran all our experiments on Android 5.0 (Nexus 4).

Security. We discuss data security risks that each of the three IoT apps pose when run on existing platforms, and find that FlowFence eliminates those risks successfully under leakage tests.

1) SmartLights: It has the potential to leak location information to attackers via the Internet. The app has Internet access for ads, and crash reporting. On FlowFence, the developer separates code that computes on location in a QM which isolates the flow: `loc → switch`, while allowing other code to use the Internet freely.

2) FaceDoor: This app can leak camera data to the Internet. We note that this app requires Internet access for core functionality—it sends a notification to the user whenever the door state changes. Therefore, under current IoT frameworks it is very easy for this app to leak camera data. FlowFence isolates the flow of camera and door state data to door locks from the flow of door state data to the Internet using two QMs, eliminating any possibility of cross-flows between the camera and the Internet. This app uses the flows: `cam → lock`, `doorstate → lock`, `doorstate → Internet`.

3) HeartRateMonitor: The app can leak images of people, plus heart rate information derived from the camera stream. However, similar to previous apps, the developer of this app too will use FlowFence support to isolate the

Name	Description	Data Security Risk without FlowFence	LoC original	LoC FlowFence	Flow Request
SmartLights [47]	Reads a location beacon and if the beacon is inside a geofence around the home, automatically turn on the lights	App can leak user location information	118	193	<code>loc → switch</code>
FaceDoor [34]	Uses a camera to recognize a face; If the face is authorized, unlock a doorlock	App can leak images of people	322	456	<code>cam → lock, doorstate → lock, doorstate → net</code>
HeartRateMonitor [67]	Uses a camera to measure heart rate and display on UI	App can leak images of people, and heart rate information	257	346	<code>cam → ui</code>

Table 2: Features of the three IoT apps ported to FlowFence. Implementing FlowFence adds 99 lines of code on average to each app (less than 140 lines per app).

flow: `cam → ui` into a QM. We note that in all apps, the QMs can return opaque handles to the pieces of code not dealing with sensitive information, where the handle can be leaked, but this is of no value to the attacker since a handle is not sensitive data.

Developer Effort. Porting apps to FlowFence requires converting pieces of code operating on sensitive data to QMs. On average, 99 lines of code were added to each app (Table 2). We note that typical IoT apps today are relatively small in size compared to, say, Android apps. The average size across 499 apps for which we have source code for SmartThings platform is 162 line of source code. Most are event-driven, receiving data from various publishers that they are authorized to at install time and then publish to various sinks, including devices or Internet. Much of the extra code deals with resolving the appropriate QMs, and creating services to communicate with FlowFence. It took a developer with no prior knowledge of the FlowFence API to port the first two apps in two 8-hour (approx.) days each, and the last app in a single day. We envision that with appropriate developer tool support, many boiler plate tasks, including QM resolution, can be automated. We note that the increase in LoC is not co-related to the original LoC of the app. Instead, there is an increase in LoC only for pieces of the original app that deals with sensitive data. Furthermore, it is our experience that refactoring an existing app requires copying logic as-is, and building QMs around it. For instance, we did not have source-code access to the Qualcomm Face Recognition SDK, but we were able to successfully port the app to FlowFence.

Porting FaceDoor. Here, we give an example of the steps involved in porting an app. First, we removed all code from the app related to camera access, because FlowFence provides a camera API that allows QMs to

take pictures, and access the corresponding bitmaps. Next, we split out face recognition operations into its own Quarantined Module— QM_{recog} , that loads the native code face recognition SDK. We modified QM_{recog} to use the Trusted API to access a camera image, an operation that causes it to be tainted with camera data. We modified the pieces of code related to manipulating a ZWave lock to instead use FlowFence-provided API for accessing door locks. We also created QM_{report} that reads the door state source and then sends a notification to the user using the Internet. These two QMs isolate the flow from camera and door state to door lock, and the flow from door state to the Internet, effectively preventing any privacy violating flow of camera data to the Internet, which would otherwise be possible with current IoT frameworks.

End-to-End Latency. We quantified the impact of FlowFence on latency for various operations in the apps that are crucial to their functionality. We measured latency as the time it takes for an app to perform one entire computational cycle. In the case of SmartLights, one cycle is the time when the beacon reports a location value, till the time the app issues an operation to manipulate a switch. We observed a latency of $160ms$ ($SD=69.9$) for SmartLights in the baseline case, and a latency of $270ms$ ($SD=96.1$) in the FlowFence case. The reason for increased latency is due to QM load time, and cross-process transfers of the location predicate value.

FaceDoor has two operations where latency matters. First, the enroll latency is the time it takes the app to extract features from a provided bitmap of a person’s face. Second, recognition latency is the time it takes the app to match a given bitmap of a person’s face to an item in the app’s database of features. We used images of our team members (6), measuring 612×816 pixels with an average

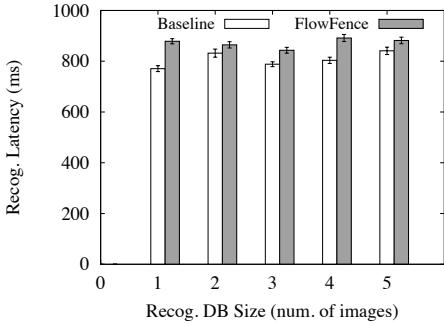


Figure 5: FaceDoor Recognition Latency (*ms*) on varying DB sizes for Baseline and FlowFence. Using FlowFence causes 5% increase in average latency.

HeartRateMonitor Metric (fps)	Baseline	FlowFence
	Avg (SD)	Avg (SD)
Throughput with no Image Processing	23.0 (0.7)	22.9 (0.7)
Throughput with Image Processing	22.9 (0.7)	22.7 (0.7)

Table 3: Throughput for HeartRateMonitor on Baseline (Stock Android) and FlowFence. FlowFence imposes little overhead on the app.

size of 290.3kB (SD=15.2).

We observed an enroll latency of 811ms (SD=37.1) in the baseline case, and 937ms (SD=60.4) for FlowFence, averaged over 50 trials. The increase in latency (15.5%) is due to QM load time, and marshaling costs for transferring bitmaps over process boundaries. While the increase in latency is well within bounds of network variations, and undetectable by user in both previous cases; it is important to recognize that most of this increase is resulted from setup time and the effect on actual processing time is much more modest. Figure 5 shows latency for face recognition, averaged over 10 trials, for Baseline, and FlowFence. We varied the recognition database size from 1 to 5 images. In each test, the last image enrolled in the database is a specific person’s face that we designated as the test face. While invoking the recognition operation, we used another image of the same test person’s face. We observe a modest, and expected increase in latency when FaceDoor runs on FlowFence. For instance, it took 882ms to successfully recognize a face in a DB of 5 images and unlock the door on FlowFence, compared to 841ms on baseline—a 4.9% increase. This latency is smaller than 100ms and thus small enough to not cause user-noticeable delays in unlocking a door once a face is recognized [13].

Throughput. Table 3 summarizes the throughput in

frames per second (fps) for HeartRateMonitor. We observed a throughput of 23.0*fps* on Stock Android for an app that read frames at maximum rate from a camera over a period of 120 seconds. We repeated the same experiment with the image processing load of heart rate detection, and observed no change in throughput. These results matched our expectations, given that the additional serialization and call latency is too low to impact the throughput of reading from the camera (camera was the bottleneck). Thus, we observed no change in the app’s abilities to derive heart rate.

6 Discussion and Limitations

Overtainting. Overtainting is difficult to avoid in taint propagation systems. FlowFence limits overtainting in two ways: (1) by not propagating taint labels from a QM to its caller—an opaque handle returned as a result of a call to a QM has an associated taint but does not cause the caller to become tainted (unless the caller is a QM that dereferences the handle), limiting the taints to QMs; and (2) a QM (and associated sandbox) is ephemeral. Since FlowFence sanitizes sandboxes if a new occupant’s taints differ from the previous occupant, reusing sandboxes does not cause overtainting. Nevertheless, FlowFence does not prevent overtainting due to poor application decomposition into QMs.

A malicious publisher can potentially overtaint a consumer by publishing overtainted data that the consumer subscribes to, leading to poison-pill attacks [37]. A plausible defense strategy is to allow a consumer to inspect an item’s taint and not proceed with a read if the item is overtainted [63]. However, this risks introducing a signaling mechanism from a high producer to a low consumer via changes to the item’s taint set. To address the attack in the context of our system. We first observe that most publishers will publish their sensor data under a known, fixed taint. The key idea is to simply require publishers to define a *taint bound* TM_c , whenever a channel c is created.⁸ If the publisher writes data with a taint set T that is not a subset of TM_c to the channel c , the write operation is denied and results in an exception; else the write is allowed. The consumer, to avoid getting overtainted, can inspect this channel’s taint bound (but not the item’s taint) before deciding to read an item from the channel. The taint bound cannot be modified, once defined, avoiding the signaling problem. A similar defense mechanism was proposed in label-based IFC systems [63, 62].

Applicability of Opacified Computation to other domains. In this work we only discussed Opacified Com-

⁸Same idea applies when creating keys, with a taint bound defined at that time for any future value associated with the key.

putation in the context of IoT frameworks (*e.g.*, FlowFence Key-Value Store and Event Channels are inspired by our IoT framework study). The basic Opacified Computation model is broadly applicable. For example, there is nothing fundamental preventing our hub from being a mobile smartphone and the app running on it being a mobile app. But, applying FlowFence to existing mobile apps is challenging because of the need to refactor apps and the libraries they use (many of the libraries access sensitive data as well as sinks). As another design point, there is no fundamental limitation that requires IoT hub software to run in a user’s home; it could well be cloud-hosted and provided as a trusted cloud-based service for supporting computations on sensitive data. Use of a cloud-based service for executing apps is not unusual—SmartThings runs all apps on its cloud, using a hub to primarily serve as a gateway for connecting devices to the cloud-based apps.

Usability of Flow Prompts. FlowFence suffers from the same limitation as all systems where users need to make security decisions, in that we cannot prevent users from approving flows that they should not. FlowFence does offer additional information during prompts since it presents flow requests with sources and sinks indicating *how* the app intends to use data, possibly leading to more informed decision-making. Flow prompts to request user permissions could be avoided if publisher policies always overrode consumer policies, with no user override allowed. But that just shifts the burden to specifying publisher policies correctly, which still may require user involvement. User education on flow policies and further user studies are likely going to be required to examine usability of flow prompts. In some IoT environments, the right to configure policies or grant overrides could be assigned to specially-trained administrators who manage flow policies on behalf of users and install apps and devices for them.

Measuring flows. Almuhamidi *et al.* performed a user study that suggests that providing metrics on frequency of use of a previously granted permission can nudge users to patch their privacy policy [6]. For example, if a user is told that an app read their location 5,398 times over a day, they may be more inclined to prevent that app from getting full access to the location. Adding support for measuring flows (both permitted and denied) to assist users in evaluating past flow permissions is part of future work.

Side Channels. A limitation of our current design is that attackers can encode sensitive data values in the time it takes for QMs to return. Such side channel techniques are primarily applicable to leaking low-bandwidth data. Nevertheless, we are investigating techniques to restrict this particular channel by making QMs return immediately, and have them execute asynchronously, thus elim-

inating the availability of fine-grain timing information in the opaque handles (as in LIO [61]). This would involve creating opaque handle dependency graphs that determine how to schedule QMs for later execution. Furthermore, timing channel leakages can be bounded using predictive techniques [72].

7 Related Work

IoT Security. Current research focuses around analyzing the security of devices [35, 27], protocols [44, 11], or platforms [26, 12]. For example, Fernandes *et al.* showed how malicious apps can steal pincodes [26]. Current IoT frameworks only offer access control but not data-flow control primitives (§2). In contrast, our work introduces, to the best of our knowledge, the first security model targeted at controlling data flows in IoT apps.

Permission Models. We observe that IoT framework permissions are modeled after smartphone permissions. There has been a large research effort at analyzing, and improving access control in smartphone frameworks [20, 49, 22, 24, 51, 50, 10, 16, 43, 68]. For instance, Enck *et al.* introduced the idea that dangerous permission combinations are indicative of possibly malicious activity [20]. Roesner *et al.* introduced User-Driven Access control where apps prompt for permissions only when they need it [51, 50]. However, permissions are fundamentally only gate-keepers. The *PlaceRaider* sensory malware abuses granted permissions and uses smartphone sensors (*e.g.*, camera) to reconstruct the 3D environment of the user for reconnaissance [64]. This malware exploits the inability of permission systems to control data usage once access is granted. The IoT fundamentally has a lot more sensitive data than a single smartphone camera, motivating the need for a security model that is capable of strictly controlling data use once apps obtain access. PiBox does offer privacy guarantees using differential-privacy algorithms after apps gain permissions, but it is primarily applicable to apps that gather aggregate statistics [43]. In contrast, FlowFence controls data flows between arbitrary types of publishers and consumers.

Label-based Information Flow Control. FlowFence builds on substantial prior work on information flow control that use labeling architectures [52, 42, 18, 71, 63, 28, 62, 41, 15, 38, 46]. For example, Flume [42] enforces flow control at the level of processes while retaining existing OS abstractions, Hails [28] presents a web framework that uses MAC to confine untrusted web apps, and COWL [63] introduces labeled compartments for JavaScript code in web apps. Although FlowFence is closely related to such systems, it also makes design choices tailored to meet the needs specific to the IoT domain. In terms of similarities, FlowFence shares the design principles of making information flow explicit, con-

trolling information flow at a higher granularity than the instruction-level, and supporting declassification. However, these systems only support producer (source) defined policies whereas FlowFence supports policies defined by both producing and consuming apps. This feature allows for more versatility in environments such as IoT, where a variety of consuming apps could request for a diverse set of flows. Our evaluation shows hows such a mix of flow policies supports real IoT apps (§5.2).

Computation on Opacified Data. Jana *et al.* built the recognizer OS abstraction and Darkly [39, 40]—systems that enable apps to compute on perceptual data while protecting the user’s privacy. These systems also use opaque handles, but they only support trusted functions operating on the raw data that handles refer to. In contrast, FlowFence supports untrusted third-party functions executing over raw data while providing flow control guarantees. Furthermore, these systems leverage characteristics of the data they are trying to protect to achieve security guarantees. For example, Darkly depends on camera streams being amenable to privacy transforms, allowing it to substitute low-fidelity data for high-fidelity data, and it depends on apps being able to tolerate the differences. However, in the general case, neither IoT data nor their apps may be amenable to such transforms. FlowFence is explicitly designed to support computation over sensitive IoT data in the general case.

Taint Tracking. Taint tracking systems [69, 19] are popular techniques for enforcing flow control that monitor data flows through programs [60]. Beyond performance issues [48], such techniques suffer from an inability to effectively handle implicit flows, and concurrency [59]. Although there are techniques to reduce computational burden [54, 65], they often require specialized hardware, not necessarily available in IoT environments. These techniques are also difficult to apply to situations where taint labels are not known *a priori* (*e.g.*, manage tainted data that is generated by apps, rather than known sources). Compared to these techniques, FlowFence adds little performance overhead. Furthermore, FlowFence does not require specialized hardware, and does not suffer from implicit flow attacks.

Static Analysis. Another class of systems such as FlowDroid [9], and Amandroid [66] use static taint tracking to enforce flow control. While these techniques do not suffer from performance issues associated with dynamic systems, they still suffer from same shortcomings associated with concurrency and implicit flows [9]. Besides static analysis techniques, there are also language-based techniques, such as JFlow [45], that require the developer to learn and use a single security-typed language. In contrast, FlowFence supports building apps using unmodified existing languages and development tools, enabling developers to quickly port their apps.

8 Conclusions

Emerging IoT programming frameworks only support permission based access control on sensitive data, making it possible for malicious apps to abuse permissions and leak data. In this work, we introduce the Opacified Computation model, and its concrete instantiation, FlowFence, which requires consumers of sensitive data to explicitly declare intended data flows. It enforces the declared flows and prevents all other flows, including implicit flows, efficiently. To achieve this, FlowFence requires developers to split their apps into: (1) A set of communicating Quarantined Modules with the unit of communication being opaque handles—taint tracked, opaque references to data that can only be dereferenced inside sandboxes; (2) Non-sensitive code that does not compute on sensitive data, but it still orchestrates execution of Quarantined Modules that compute on sensitive data. We ported three IoT apps to FlowFence, each requiring less than 140 additional lines of code. Latency and throughput measurements of crucial operations of the ported apps indicate that FlowFence adds little overhead. For instance, we observed a 4.9% latency increase to recognize a face in a door controller app.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Deian Stefan, for their insightful feedback on our work. We thank Kevin Borders, Kevin Eykholt, and Jaeyeon Jung for providing feedback on earlier drafts. This research is supported in part by NSF grant CNS-1318722 and by a generous gift from General Motors. Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). His work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), the Italian MIUR-PRIN TENACE Project (agreement 20103P34XC), and by the projects “Tackling Mobile Malware with Innovative Machine Learning Techniques,” “Physical-Layer Security for Wireless Communication,” and “Content Centric Networking: Security and Privacy Issues” funded by the University of Padua. Any opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

References

- [1] Android auto. <https://www.android.com/auto/>. Accessed: May 2016.
- [2] Android wear. <https://www.android.com/wear/>. Accessed: May 2016.

- [3] Logitech harmony hub. <http://www.logitech.com/en-us/product/harmony-hub>. Accessed: May 2016.
- [4] Samsung SmartThings Home Automation. <http://www.smartthings.com/>. Accessed: Oct 2015.
- [5] Vera Smart Home Controller. <http://getvera.com/controllers/vera3/>. Accessed: Oct 2015.
- [6] ALMUHIMEDI, H., SCHAUB, F., SADEH, N., ADJERID, I., ACQUISTI, A., GLUCK, J., CRANOR, L. F., AND AGARWAL, Y. Your Location Has Been Shared 5,398 Times!: A Field Study on Mobile App Privacy Nudging. In *ACM Conference on Human Factors in Computing Systems (CHI)* (2015).
- [7] APPLE. Apple TV Memory Specifications. https://developer.apple.com/library/tvos/documentation/General/Conceptual/AppleTV_PG/index.html#/apple_ref/doc/uid/TP40015241-CH12-SW1. Accessed: June 2016.
- [8] APPLE. HomeKit. <http://www.apple.com/ios/homekit/>. Accessed: Oct 2015.
- [9] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND McDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM symposium on Programming Language Design and Implementation (PLDI)* (2014).
- [10] BACKES, M., BUGIEL, S., AND GERLING, S. Scippa: System-centric ipc provenance on android. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014).
- [11] BEHRANG FOULADI AND SAHAND GHANOUN. Honey, I'm Home!! Hacking ZWave Home Automation Systems. Black Hat USA, 2013.
- [12] BUSOLD, C., HEUSER, S., RIOS, J., SADEGHI, A.-R., AND ASOKAN, N. Smart and secure cross-device apps for the internet of advanced things. In *Financial Cryptography and Data Security (FC)* (2015).
- [13] CARD, S. K., ROBERTSON, G. G., AND MACKINLAY, J. D. The information visualizer, an information workspace. In *SIGCHI Conference on Human factors in computing systems* (1991).
- [14] CARLINI, N., FELT, A. P., AND WAGNER, D. An evaluation of the google chrome extension security architecture. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (2012).
- [15] CHENG, W., PORTS, D. R., SCHULTZ, D., POPIC, V., BLANKSTEIN, A., COWLING, J., CURTIS, D., SHRIRA, L., AND LISKOV, B. Abstractions for usable information flow control in aeolus. In *USENIX ATC* (2012).
- [16] CONTI, M., CRISPÓ, B., FERNANDES, E., AND ZHANIAROVICH, Y. Crêpe: A system for enforcing fine-grained context-related policies on android. *TIFS* (2012).
- [17] DENNING, T., KOHNO, T., AND LEVY, H. M. Computer security and the modern home. *Communications of ACM* (2013).
- [18] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. In *SOSP* (2005).
- [19] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., McDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010).
- [20] ENCK, W., ONGTANG, M., AND McDANIEL, P. On lightweight mobile phone application certification. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2009).
- [21] ERNST, M. D., JUST, R., MILLSTEIN, S., DIETL, W., PERNSTEINER, S., ROESNER, F., KOSCHER, K., BARROS, P. B., BHORASKAR, R., HAN, S., VINES, P., AND WU, E. X. Collaborative verification of information flow for a high-assurance app store. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2014).
- [22] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2011).
- [23] FELT, A. P., EGELMAN, S., FINFTER, M., AKHAWE, D., AND WAGNER, D. How to ask for permission. In *USENIX Conference on Hot Topics in Security (HotSec)* (2012).
- [24] FELT, A. P., EGELMAN, S., AND WAGNER, D. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (2012).
- [25] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS)* (2012), Symposium On Usable Privacy and Security (SOUPS).
- [26] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security analysis of emerging smart home applications. In *IEEE Symposium on Security and Privacy (S&P)* (2016).
- [27] FISHER, D. Pair of Bugs Open Honeywell Home Controllers Up to Easy Hacks. <https://threatpost.com/pair-of-bugs-open-honeywell-home-controllers-up-to-easy-hacks/113965/>. Accessed: Oct 2015.
- [28] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J. C., AND RUSSO, A. Hails: Protecting data privacy in untrusted web applications. In *OSDI* (2012).
- [29] GOOGLE. Project Brillo. <https://developers.google.com/brillo/>. Accessed: Oct 2015.
- [30] GOOGLE. Project Weave. <https://developers.google.com/weave/>. Accessed: Oct 2015.
- [31] GOOGLE ANDROID. Requesting Permissions at Runtime. <http://developer.android.com/training/permissions/requesting.html>. Accessed: Feb 2016.
- [32] GOOGLE DEVELOPERS. Google Fit Developer Documentation. <https://developers.google.com/fit/>. Accessed: Feb 2016.
- [33] GOOGLE NEST. How much bandwidth will Nest cam use? <https://nest.com/support/article/How-much-bandwidth-will-Nest-Cam-use>. Accessed: June 2016.
- [34] HACHMAN, M. Want to unlock your door with your face? Windows 10 for IoT Core promises to do just that. <http://www.pcworld.com/article/2962330/internet-of-things/want-to-unlock-your-door-with-your-face-windows-10-for-iot-core-promises-to-do-just-that.html>. Accessed: Feb 2016.
- [35] HESSELDALH, A. A Hacker's-Eye View of the Internet of Things. <http://recode.net/2015/04/07/a-hackers-eye-view-of-the-internet-of-things/>. Accessed: Oct 2015.
- [36] HEULE, S., RIFKIN, D., RUSSO, A., AND STEFAN, D. The most dangerous code in the browser. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015), USENIX Association.
- [37] HRITCU, C., GREENBERG, M., KAREL, B., PIERCE, B. C., AND MORRISETT, G. All your ifcexception are belong to us. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), SP '13.

- [38] HRITCU, C., GREENBERG, M., KAREL, B., PIERCE, B. C., AND MORRISETT, G. All your ifcexception are belong to us. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE.
- [39] JANA, S., MOLNAR, D., MOSHCHUK, A., DUNN, A., LIVSHITS, B., WANG, H. J., AND OFEK, E. Enabling fine-grained permissions for augmented reality applications with recognizers. In *USENIX Security Symposium* (2013).
- [40] JANA, S., NARAYANAN, A., AND SHMATIKOV, V. A Scanner Darkly: Protecting User Privacy from Perceptual Applications. In *IEEE Symposium on Security and Privacy (S&P)* (2013).
- [41] JIA, L., ALJURAI DAN, J., FRAGKAKI, E., BAUER, L., STROUCKEN, M., FUKUSHIMA, K., KIYOMOTO, S., AND MIYAKE, Y. Run-time enforcement of information-flow properties on android. In *European Symposium on Research in Computer Security* (2013).
- [42] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard os abstractions. In *SOSP* (2007).
- [43] LEE, S., WONG, E. L., GOEL, D., DAHLIN, M., AND SHMATIKOV, V. box: A platform for privacy-preserving apps. In *NSDI* (2013).
- [44] LOMAS, N. Critical Flaw identified In ZigBee Smart Home Devices. <http://techcrunch.com/2015/08/07/critical-flaw-identified-in-zigbee-smart-home-devices/>. Accessed: Oct 2015.
- [45] MYERS, A. C. Jflow: Practical mostly-static information flow control. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (1999).
- [46] NADKARNI, A., AND ENCK, W. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM.
- [47] PANSARASA, J. Lights-After-Dark SmartThings App. <https://github.com/jpansarasa/SmartThings/blob/master/smartsapps/elasticdev/lights-after-dark/src/lights-after-dark.groovy>. Accessed: Feb 2016.
- [48] PAUPORE, J., FERNANDES, E., PRAKASH, A., ROY, S., AND OU, X. Practical always-on taint tracking on mobile devices. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)* (2015).
- [49] RAHMATI, A., AND MADHYASTHA, H. V. Context-specific access control: Conforming permissions with user expectations. In *ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)* (2015).
- [50] ROESNER, F., AND KOHNO, T. Securing embedded user interfaces: Android and beyond. In *USENIX Security Symposium* (2013).
- [51] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., AND COWAN, C. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE S&P* (2012).
- [52] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: Practical fine-grained decentralized information flow control. In *PLDI* (2009).
- [53] RUSSELLO, G., CONTI, M., CRISPO, B., AND FERNANDES, E. Moses: Supporting operation modes on smartphones. In *ACM Symposium on Access Control Models and Technologies (SACMAT)* (2012).
- [54] RUWASE, O., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., CHEN, S., KOZUCH, M., AND RYAN, M. Parallelizing dynamic information flow tracking. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures* (2008).
- [55] SAMSUNG. SmartThings. <http://www.smartthings.com/>. Accessed: Nov 2015.
- [56] SAMSUNG SMARTTHINGS. Samsung SmartThings Memory Specifications. <https://community.smarththings.com/t/the-next-generation-of-smarththings-is-here/21521>. Accessed: June 2016.
- [57] SAMSUNG SMARTTHINGS. SmartThings Capabilities Reference. <http://docs.smarththings.com/en/latest/capabilities-reference.html>. Accessed: Feb 2016.
- [58] SAMSUNG SMARTTHINGS. What happens if the power goes out or I lose my internet connection? <https://support.smarththings.com/hc/en-us/articles/205956960-What-happens-if-the-power-goes-out-or-I-lose-my-internet-connection->. Accessed: May 2016.
- [59] SARWAR, G., MEHANI, O., BORELI, R., AND KAAFAR, M. A. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *International Conference on Security and Cryptography (SECRYPT)* (2013).
- [60] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)* (2010).
- [61] STEFAN, D., RUSSO, A., BUIRAS, P., LEVY, A., MITCHELL, J. C., AND MAZIÈRES, D. Addressing covert termination and timing channels in concurrent information flow systems. In *ACM SIGPLAN Notices* (2012).
- [62] STEFAN, D., RUSSO, A., MITCHELL, J. C., AND MAZIÈRES, D. Flexible dynamic information flow control in Haskell. In *Haskell Symposium* (September 2011), ACM SIGPLAN.
- [63] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIÈRES, D. Protecting users by confining javascript with cowl. In *OSDI* (2014).
- [64] TEMPLEMAN, R., RAHMAN, Z., CRANDALL, D., AND KAPADIA, A. PlaceRaider: Virtual theft in physical spaces with smartphones. In *ISOC Network and Distributed System Security Symposium (NDSS)* (2013).
- [65] VACHHARAJANI, N., BRIDGES, M. J., CHANG, J., RANGAN, R., OTTONI, G., BLOME, J. A., REIS, G. A., VACHHARAJANI, M., AND AUGUST, D. I. Rifle: An architectural framework for user-centric information-flow security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on* (2004).
- [66] WEI, F., ROY, S., OU, X., AND ROBBY. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2014).
- [67] WETHERELL, J. Android Heart Rate Monitor App. <https://github.com/phishman3579/android-heart-rate-monitor>. Accessed: Feb 2016.
- [68] XU, Y., HUNT, T., KWON, Y., GEORGIEV, M., SHMATIKOV, V., AND WITCHEL, E. Earp: Principled storage, sharing, and protection for mobile apps. In *NSDI* (2016).
- [69] XU, Y., AND WITCHEL, E. Maxoid: Transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM.
- [70] YOON, M.-K., SALAJEGHEH, N., CHEN, Y., AND CHRISTODORESCU, M. Pift: Predictive information flow tracking. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems* (2016).

- [71] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *OSDI* (2006).
- [72] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Predictive mitigation of timing channels in interactive systems. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2011).
- [73] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *IEEE S&P* (2012).

Appendix A: FlowFence API

We summarize the object-oriented FlowFence API for developers in Table 4. There are two kinds of API: QM-management, and Within-QM. Developers use the QM-management API to request loading QMs into sandboxes, making QM calls, and receiving opaque handles as return values. The primary data types are: $QM <T>$, and $Handle$. The former data type represents a reference to a loaded QM. The latter data type represents an opaque handle, that FlowFence creates as a return value of a QM. Developers use `resolveCtor`, or `resolveM` to load a specific QM into a sandbox (FlowFence automatically manages sandboxes), and receive a reference to the loaded QM. Then, developers specify the string name of a QM method to execute.

The Within-QM API is available to QMs while they are executing within a sandbox. Currently, FlowFence has two data types available for QMs. *KVStore* offers ways to get and put values in the Key-Value store. The Trusted API offers facilities like network communication, logging, and smart home control (our prototype has a bridge to SmartThings).

QM-management Data Types and API	Semantics
<code>Handle</code>	An opaque handle. Data is stored in the Trusted Service, with its taint labels.
$QM <T>$	A reference to a QM of type T, on which developers can issue method calls.
$QM <T> \text{ ctor} = \text{resolveCtor}(T)$	Resolve the constructor for QM T, and return a reference to it.
$QM <T> m = \text{resolveM}(\text{retType}, T, \text{methStr}, [\text{paramTypes}])$	Resolve an instance/static method of a QM, loading the QM into a sandbox if necessary.
$Handle \text{ ret} = QM <T>.\text{call}([\text{argList}])$	Call a method on a loaded QM, and return an opaque handle as the result.
<code>subscribeEventChannel(appID, $QM <T>$)</code>	Subscribe to a channel for updates, and register a QM to be executed automatically whenever new data is placed on the channel.
Within-QM Data Types and API	Semantics
<code>KVStore</code>	Provides methods to interact with the Key-Value Store.
<code>KVStore kvs = getKVStore(appID, name)</code>	Get a reference to a named KVStore.
<code>kvs.put<T>(key, value, taint_label)</code>	Put a (key, value) pair into the KVStore along with a taint label, where T can be a basic type such as Int, Float, or a serializable type. Any existing taint of the calling QM will be automatically associated with the value's final set of taint labels.
<code>T value = kvs.get<T>(key)</code>	Get the value of type T corresponding to specified key, and taint the QM with the appropriate set of taint labels.
<code>getTrustedAPI(apiName).invoke([params])</code>	Call a Trusted API method to declassify sensitive data.
<code>getChannel(chanName).fireEvent(taint_label, [params])</code>	Fire an event with parameters, specifying taint label. Any existing taint labels of the calling QM will be added automatically.

Table 4: FlowFence API Summary. QM-management data types and API is only available to the untrusted portion of an app that does not operate with sensitive data. The Within-QM data types and API is available only to QMs.

ARMageddon: Cache Attacks on Mobile Devices

Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard
Graz University of Technology, Austria

Abstract

In the last 10 years, cache attacks on Intel x86 CPUs have gained increasing attention among the scientific community and powerful techniques to exploit cache side channels have been developed. However, modern smartphones use one or more multi-core ARM CPUs that have a different cache organization and instruction set than Intel x86 CPUs. So far, no cross-core cache attacks have been demonstrated on non-rooted Android smartphones. In this work, we demonstrate how to solve key challenges to perform the most powerful cross-core cache attacks *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, and *Flush+Flush* on non-rooted ARM-based devices without any privileges. Based on our techniques, we demonstrate covert channels that outperform state-of-the-art covert channels on Android by several orders of magnitude. Moreover, we present attacks to monitor tap and swipe events as well as keystrokes, and even derive the lengths of words entered on the touchscreen. Eventually, we are the first to attack cryptographic primitives implemented in Java. Our attacks work across CPUs and can even monitor cache activity in the ARM TrustZone from the normal world. The techniques we present can be used to attack hundreds of millions of Android devices.

1 Introduction

Cache attacks represent a powerful means of exploiting the different access times within the memory hierarchy of modern system architectures. Until recently, these attacks explicitly targeted cryptographic implementations, for instance, by means of cache timing attacks [9] or the well-known *Evict+Time* and *Prime+Probe* techniques [43]. The seminal paper by Yarom and Falkner [60] introduced the so-called *Flush+Reload* attack, which allows an attacker to infer which specific parts of a binary are accessed by a victim program with an unprecedented accuracy and probing frequency. Recently, Gruss et al. [19] demonstrated

the possibility to use *Flush+Reload* to automatically exploit cache-based side channels via cache template attacks on Intel platforms. *Flush+Reload* does not only allow for efficient attacks against cryptographic implementations [8, 26, 56], but also to infer keystroke information and even to build keyloggers on Intel platforms [19]. In contrast to attacks on cryptographic algorithms, which are typically triggered multiple times, these attacks require a significantly higher accuracy as an attacker has only one single chance to observe a user input event.

Although a few publications about cache attacks on AES T-table implementations on mobile devices exist [10, 50–52, 57], the more efficient cross-core attack techniques *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, and *Flush+Flush* [18] have not been applied on smartphones. In fact, there was reasonable doubt [60] whether these cross-core attacks can be mounted on ARM-based devices at all. In this work, we demonstrate that these attack techniques are applicable on ARM-based devices by solving the following key challenges systematically:

1. *Last-level caches are not inclusive on ARM and thus cross-core attacks cannot rely on this property.* Indeed, existing cross-core attacks exploit the inclusiveness of shared last-level caches [18, 19, 22, 24, 35, 37, 38, 42, 60] and, thus, no cross-core attacks have been demonstrated on ARM so far. We present an approach that exploits coherence protocols and L1-to-L2 transfers to make these attacks applicable on mobile devices with non-inclusive shared last-level caches, irrespective of the cache organization.¹
2. *Most modern smartphones have multiple CPUs that do not share a cache.* However, cache coherence protocols allow CPUs to fetch cache lines from remote cores faster than from the main memory. We utilize this property to mount both cross-core and cross-CPU attacks.

¹Simultaneously to our work on ARM, Irazoqui et al. [25] developed a technique to exploit cache coherence protocols on AMD x86 CPUs and mounted the first cross-CPU cache attack.

3. *Except ARMv8-A CPUs, ARM processors do not support a flush instruction.* In these cases, a fast eviction strategy must be applied for high-frequency measurements. As existing eviction strategies are too slow, we analyze more than 4 200 eviction strategies for our test devices, based on Rowhammer attack techniques [17].
4. *ARM CPUs use a pseudo-random replacement policy* to decide which cache line to replace within a cache set. This introduces additional noise even for robust time-driven cache attacks [50, 52]. For the same reason, *Prime+Probe* has been an open challenge [51] on ARM, as an attacker needs to predict which cache line will be replaced first and wrong predictions destroy measurements. We design re-access loops that interlock with a cache eviction strategy to reduce the effect of wrong predictions.
5. *Cycle-accurate timings require root access on ARM* [3] and alternatives have not been evaluated so far. We evaluate different timing sources and show that cache attacks can be mounted in any case.

Based on these building blocks, we demonstrate practical and highly efficient cache attacks on ARM.² We do not restrict our investigations to cryptographic implementations but also consider cache attacks as a means to infer other sensitive information—such as inter-keystroke timings or the length of a swipe action—requiring a significantly higher measurement accuracy. Besides these generic attacks, we also demonstrate that cache attacks can be used to monitor cache activity caused within the ARM TrustZone from the normal world. Nevertheless, we do not aim to exhaustively list possible exploits or find new attack vectors on cryptographic algorithms. Instead, we aim to demonstrate the immense attack potential of the presented cross-core and cross-CPU attacks on ARM-based mobile devices based on well-studied attack vectors. Our work allows to apply existing attacks to millions of off-the-shelf Android devices without any privileges. Furthermore, our investigations show that Android still employs vulnerable AES T-table implementations.

Contributions. The contributions of this work are:

- We demonstrate the applicability of highly efficient cache attacks like *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, and *Flush+Flush* on ARM.
- Our attacks work irrespective of the actual cache organization and, thus, are the first last-level cache attacks that can be applied cross-core and also cross-CPU on off-the-shelf ARM-based devices. More specifically, our attacks work against last-

²Source code for ARMageddon attack examples can be found at <https://github.com/IAIK/armageddon>.

level caches that are instruction-inclusive and data-non-inclusive as well as caches that are instruction-non-inclusive and data-inclusive.

- Our cache-based covert channel outperforms all existing covert channels on Android by several orders of magnitude.
- We demonstrate the power of these attacks by attacking cryptographic implementations and by inferring more fine-grained information like keystrokes and swipe actions on the touchscreen.

Outline. The remainder of this paper is structured as follows. In Section 2, we provide information on background and related work. Section 3 describes the techniques that are the building blocks for our attacks. In Section 4, we demonstrate and evaluate fast cross-core and cross-CPU covert channels on Android. In Section 5, we demonstrate cache template attacks on user input events. In Section 6, we present attacks on cryptographic implementations used in practice as well the possibility to observe cache activity of cryptographic computations within the TrustZone. We discuss countermeasures in Section 7 and conclude this work in Section 8.

2 Background and Related Work

In this section, we provide the required preliminaries and discuss related work in the context of cache attacks.

2.1 CPU Caches

Today’s CPU performance is influenced not only by the clock frequency but also by the latency of instructions, operand fetches, and other interactions with internal and external devices. In order to overcome the latency of system memory accesses, CPUs employ caches to buffer frequently used data in small and fast internal memories.

Modern caches organize cache lines in multiple sets, which is also known as set-associative caches. Each memory address maps to one of these cache sets and addresses that map to the same cache set are considered congruent. Congruent addresses compete for cache lines within the same set and a predefined replacement policy determines which cache line is replaced. For instance, the last generations of Intel CPUs employ an undocumented variant of least-recently used (LRU) replacement policy [17]. ARM processors use a pseudo-LRU replacement policy for the L1 cache and they support two different cache replacement policies for L2 caches, namely round-robin and pseudo-random replacement policy. In practice, however, only the pseudo-random replacement policy is used due to performance reasons. Switching the cache replacement policy is only possible in privi-

leged mode. The implementation details for the pseudo-random policy are not documented.

CPU caches can either be virtually indexed or physically indexed, which determines whether the index is derived from the virtual or physical address. A so-called tag uniquely identifies the address that is cached within a specific cache line. Although this tag can also be based on the virtual or physical address, most modern caches use physical tags because they can be computed simultaneously while locating the cache set. ARM typically uses physically indexed, physically tagged L2 caches.

CPUs have multiple cache levels, with the lower levels being faster and smaller than the higher levels. ARM processors typically have two levels of cache. If all cache lines from lower levels are also stored in a higher-level cache, the higher-level cache is called *inclusive*. If a cache line can only reside in one of the cache levels at any point in time, the caches are called *exclusive*. If the cache is neither inclusive nor exclusive, it is called *non-inclusive*. The last-level cache is often shared among all cores to enhance the performance upon transitioning threads between cores and to simplify cross-core cache lookups. However, with shared last-level caches, one core can (intentionally) influence the cache content of all other cores. This represents the basis for cache attacks like *Flush+Reload* [60].

In order to keep caches of multiple CPU cores or CPUs in a coherent state, so-called coherence protocols are employed. However, coherence protocols also introduce exploitable timing effects, which has recently been exploited by Irazoqui et al. [25] on x86 CPUs.

In this paper, we demonstrate attacks on three smartphones as listed in Table 1. The Krait 400 is an ARMv7-A CPU, the other two processors are ARMv8-A CPUs. However, the stock Android of the Alcatel One Touch Pop 2 is compiled for an ARMv7-A instruction set and thus ARMv8-A instructions are not used. We generically refer to ARMv7-A and ARMv8-A as “ARM architecture” throughout this paper. All devices have a shared L2 cache. On the Samsung Galaxy S6, the flush instruction is unlocked by default, which means that it is available in userspace. Furthermore, all devices employ a cache coherence protocol between cores and on the Samsung Galaxy S6 even between the two CPUs [6].

2.2 Shared Memory

Read-only shared memory can be used as a means of memory usage optimization. In case of shared libraries it reduces the memory footprint and enhances the speed by lowering cache contention. The operating system implements this behavior by mapping the same physical memory into the address space of each process. As this memory sharing mechanism is independent of how a file was

opened or accessed, an attacker can map a binary to have read-only shared memory with a victim program. A similar effect is caused by content-based page deduplication where physical pages with identical content are merged.

Android applications are usually written in Java and, thus, contain self-modifying code or just-in-time compiled code. This code would typically not be shared. Since Android version 4.4 the Dalvik VM was gradually replaced by the Android Runtime (ART). With ART, Java byte code is compiled to native code binaries [1] and thus can be shared too.

2.3 Cache Attacks

Initially, cache timing attacks were performed on cryptographic algorithms [9, 30, 31, 40, 41, 44, 55]. For example, Bernstein [9] exploited the total execution time of AES T-table implementations. More fine-grained exploitations of memory accesses to the CPU cache have been proposed by Percival [45] and Osvik et al. [43]. More specifically, Osvik et al. formalized two concepts, namely *Evict+Time* and *Prime+Probe*, to determine which specific cache sets were accessed by a victim program. Both approaches consist of three basic steps.

Evict+Time:

1. Measure execution time of victim program.
2. Evict a specific cache set.
3. Measure execution time of victim program again.

Prime+Probe:

1. Occupy specific cache sets.
2. Victim program is scheduled.
3. Determine which cache sets are still occupied.

Both approaches allow an adversary to determine which cache sets are used during the victim’s computations and have been exploited to attack cryptographic implementations [24, 35, 43, 54] and to build cross-VM covert channels [37]. Yarom and Falkner [60] proposed *Flush+Reload*, a significantly more fine-grained attack that exploits three fundamental concepts of modern system architectures. First, the availability of shared memory between the victim process and the adversary. Second, last-level caches are typically shared among all cores. Third, Intel platforms use inclusive last-level caches, meaning that the eviction of information from the last-level cache leads to the eviction of this data from all lower-level caches of other cores, which allows any program to evict data from other programs on other cores. While the basic idea of this attack has been proposed by Gullasch et al. [21], Yarom and Falkner extended this idea to shared last-level caches, allowing cross-core attacks. *Flush+Reload* works as follows.

Flush+Reload:

1. Map binary (e.g., shared object) into address space.
2. Flush a cache line (code or data) from the cache.

Table 1: Test devices used in this paper.

Device	SoC	CPU (cores)	L1 caches	L2 cache	Inclusiveness
OnePlus One	Qualcomm Snapdragon 801	Krait 400 (2) 2.5 GHz	2× 16 KB, 4-way, 64 sets	2 048 KB, 8-way, 2 048 sets	non-inclusive
Alcatel One Touch Pop 2	Qualcomm Snapdragon 410	Cortex-A53 (4) 1.2 GHz	4× 32 KB, 4-way, 128 sets	512 KB, 16-way, 512 sets	instruction-inclusive, data-non-inclusive
Samsung Galaxy S6	Samsung Exynos 7 Octa 7420	Cortex-A57 (4) 1.5 GHz 2.1 GHz	4× 32 KB, 4-way, 128 sets 2× 32 KB, 2-way, 256 sets	256 KB, 16-way, 256 sets 2 048 KB, 16-way, 2 048 sets	instruction-inclusive, data-non-inclusive instruction-non-inclusive, data-inclusive

3. Schedule the victim program.
4. Check if the corresponding line from step 2 has been loaded by the victim program.

Thereby, *Flush+Reload* allows an attacker to determine which specific instructions are executed and also which specific data is accessed by the victim program. Thus, rather fine-grained attacks are possible and have already been demonstrated against cryptographic implementations [22, 27, 28]. Furthermore, Gruss et al. [19] demonstrated the possibility to automatically exploit cache-based side-channel information based on the *Flush+Reload* approach. Besides attacking cryptographic implementations like AES T-table implementations, they showed how to infer keystroke information and even how to build a keylogger by exploiting the cache side channel. Similarly, Oren et al. [42] demonstrated the possibility to exploit cache attacks on Intel platforms from JavaScript and showed how to infer visited websites and how to track the user’s mouse activity.

Gruss et al. [19] proposed the *Evict+Reload* technique that replaces the flush instruction in *Flush+Reload* by eviction. While it has no practical application on x86 CPUs, we show that it can be used on ARM CPUs. Recently, *Flush+Flush* [18] has been proposed. Unlike other techniques, it does not perform any memory access but relies on the timing of the flush instruction to determine whether a line has been loaded by a victim. We show that the execution time of the ARMv8-A flush instruction also depends on whether or not data is cached and, thus, can be used to implement this attack.

While the attacks discussed above have been proposed and investigated for Intel processors, the same attacks were considered not applicable to modern smartphones due to differences in the instruction set, the cache organization [60], and in the multi-core and multi-CPU architecture. Thus, only same-core cache attacks have been demonstrated on smartphones so far. For instance, Weïß et al. [57] investigated Bernstein’s cache-timing attack [9] on a Beagleboard employing an ARM Cortex-A8 processor. Later on, Weïß et al. [58] investigated this timing attack in a multi-core setting on a development

board. As Weïß et al. [57] claimed that noise makes the attack difficult, Spreitzer and Plos [52] investigated the applicability of Bernstein’s cache-timing attack on different ARM Cortex-A8 and ARM Cortex-A9 smartphones running Android. Both investigations [52, 57] confirmed that timing information is leaking, but the attack takes several hours due to the high number of measurement samples that are required, *i.e.*, about 2^{30} AES encryptions. Later on, Spreitzer and Gérard [50] improved upon these results and managed to reduce the key space to a complexity which is practically relevant.

Besides Bernstein’s attack, another attack against AES T-table implementations has been proposed by Bogdanov et al. [10], who exploited so-called wide collisions on an ARM9 microprocessor. In addition, power analysis attacks [13] and electromagnetic emanations [14] have been used to visualize cache accesses during AES computations on ARM microprocessors. Furthermore, Spreitzer and Plos [51] implemented *Evict+Time* [43] in order to attack an AES T-table implementation on Android-based smartphones. However, so far only cache attacks against AES T-table implementations have been considered on smartphone platforms and none of the recent advances have been demonstrated on mobile devices.

3 ARMageddon Attack Techniques

We consider a scenario where an adversary attacks a smartphone user by means of a malicious application. This application *does not require any permission* and, most importantly, it can be executed in unprivileged userspace and *does not require a rooted device*. As our attack techniques do not exploit specific vulnerabilities of Android versions, they work on stock Android ROMs as well as customized ROMs in use today.

3.1 Defeating the Cache Organization

In this section, we tackle the aforementioned challenges 1 and 2, *i.e.*, the last-level cache is not inclusive and multiple processors do not necessarily share a cache level.

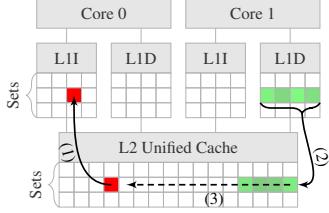


Figure 1: Cross-core instruction cache eviction through data accesses.

When it comes to caches, ARM CPUs are very heterogeneous compared to Intel CPUs. For example, whether or not a CPU has a second-level cache can be decided by the manufacturer. Nevertheless, the last-level cache on ARM devices is usually shared among all cores and it can have different inclusiveness properties for instructions and data. Due to cache coherence, shared memory is kept in a coherent state across cores and CPUs. This is of importance when measuring timing differences between cache accesses and memory accesses (cache misses), as fast remote-cache accesses are performed instead of slow memory accesses [6]. In case of a non-coherent cache, a cross-core attack is not possible but an attacker can run the spy process on all cores simultaneously and thus fall back to a same-core attack. However, we observed that caches are coherent on all our test devices.

To perform a cross-core attack we load enough data into the cache to fully evict the corresponding last-level cache set. Thereby, we exploit that we can fill the last-level cache directly or indirectly depending on the cache organization. On the Alcatel One Touch Pop 2, the last-level cache is instruction-inclusive and thus we can evict instructions from the local caches of the other core. Figure 1 illustrates such an eviction. In step 1, an instruction is allocated to the last-level cache and the instruction cache of one core. In step 2, a process fills its core's data cache, thereby evicting cache lines into the last-level cache. In step 3, the process has filled the last-level cache set using only data accesses and thereby evicts the instructions from instruction caches of other cores as well.

We access cache lines multiple times to perform transfers between L1 and L2 cache. Thus, more and more addresses used for eviction are cached in either L1 or L2. As ARM CPUs typically have L1 caches with a very low associativity, the probability of eviction to L2 through other system activity is high. Using an eviction strategy that performs frequent transfers between L1 and L2 increases this probability further. Thus, this approach also works for other cache organizations to perform cross-core and cross-CPU cache attacks. Due to the cache coherence protocol between the CPU cores [6, 33], remote-core fetches are faster than memory accesses and thus can be distinguished from cache misses. For instance,

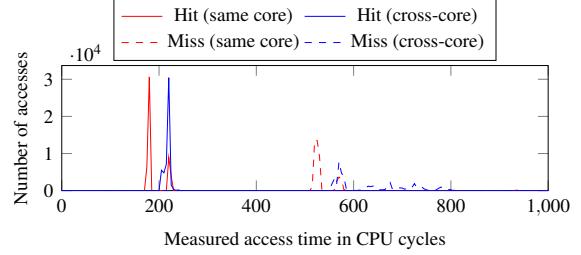


Figure 2: Histograms of cache hits and cache misses measured same-core and cross-core on the OnePlus One.

Figure 2 shows the cache hit and miss histogram on the OnePlus One. The cross-core access introduces a latency of 40 CPU cycles on average. However, cache misses take more than 500 CPU cycles on average. Thus, cache hits and misses are clearly distinguishable based on a single threshold value.

3.2 Fast Cache Eviction

In this section, we tackle the aforementioned challenges 3 and 4, *i.e.*, not all ARM processors support a flush instruction, and the replacement policy is pseudo-random.

There are two options to evict cache lines: (1) the flush instruction or (2) evict data with memory accesses to congruent addresses, *i.e.*, addresses that map to the same cache set. As the flush instruction is only available on the Samsung Galaxy S6, we need to rely on eviction strategies for the other devices and, therefore, to defeat the replacement policy. The L1 cache in Cortex-A53 and Cortex-A57 has a very small number of ways and employs a least-recently used (LRU) replacement policy [5]. However, for a full cache eviction, we also have to evict cache lines from the L2 cache, which uses a pseudo-random replacement policy.

Eviction strategies. Previous approaches to evict data on Intel x86 platforms either have too much overhead [23] or are only applicable to caches implementing an LRU replacement policy [35, 37, 42]. Spreitzer and Plos [51] proposed an eviction strategy for ARMv7-A CPUs that requires to access more addresses than there are cache lines per cache set, due to the pseudo-random replacement policy. Recently, Gruss et al. [17] demonstrated how to automatically find fast eviction strategies on Intel x86 architectures. We show that their algorithm is applicable to ARM CPUs as well. Thereby, we establish eviction strategies in an automated way and significantly reduce the overhead compared to [51]. We evaluated more than 4 200 access patterns on our smartphones and identified the best eviction strategies. Even though the cache employs a random replace-

Table 2: Different eviction strategies on the Krait 400.

<i>N</i>	<i>A</i>	<i>D</i>	Cycles	Eviction rate
-	-	-	549	100.00%
11	2	2	1 578	100.00%
12	1	3	2 094	100.00%
13	1	5	2 213	100.00%
16	1	1	3 026	100.00%
24	1	1	4 371	100.00%
13	1	2	2 372	99.58%
11	1	3	1 608	80.94%
11	4	1	1 948	58.93%
10	2	2	1 275	51.12%

Table 3: Different eviction strategies on the Cortex-A53.

<i>N</i>	<i>A</i>	<i>D</i>	Cycles	Eviction rate
-	-	-	767	100.00%
23	2	5	6 209	100.00%
23	4	6	16 912	100.00%
22	1	6	5 101	99.99%
21	1	6	4 275	99.93%
20	4	6	13 265	99.44%
800	1	1	142 876	99.10%
200	1	1	33 110	96.04%
100	1	1	15 493	89.77%
48	1	1	6 517	70.78%

ment policy, average eviction rate and average execution time are reproducible. Eviction sets are computed based on physical addresses, which can be retrieved via `/proc/self/pagemap` as current Android versions allow access to these mappings to any unprivileged app without any permissions. Thus, eviction patterns and eviction sets can be efficiently computed.

We applied the algorithm of Gruss et al. [17] to a set of physically congruent addresses. Table 2 summarizes different eviction strategies, *i.e.*, loop parameters, for the Krait 400. N denotes the total eviction set size (length of the loop), A denotes the shift offset (loop increment) to be applied after each round, and D denotes the number of memory accesses in each iteration (loop body). The column *cycles* states the average execution time in CPU cycles over 1 million evictions and the last column denotes the average eviction rate. The first line in Table 2 shows the average execution time and the average eviction rate for the privileged flush instruction, which gives the best result in terms of average execution time (549 CPU cycles). We evaluated 1 863 different strategies and our best identified eviction strategy ($N = 11$, $A = 2$, $D = 2$) also achieves an average eviction rate of 100% but takes 1 578 CPU cycles. Although a strategy accessing every address in the eviction set only once ($A = 1$, $D = 1$, also called LRU eviction) performs significantly fewer memory accesses, it consumes more CPU cycles. For an average eviction rate of 100%, LRU eviction requires an eviction set size of at least 16. The average execution time then is 3 026 CPU cycles. Considering the eviction strategy used in [51] that takes 4 371 CPU cycles, clearly demonstrates the advantage of our optimized eviction strategy that takes only 1 578 CPU cycles.

We performed the same evaluation with 2 295 different strategies on the ARM Cortex-A53 in our Alcatel One Touch Pop 2 test system and summarize them in Table 3. For the best strategy we found ($N = 21$, $A = 1$, $D = 6$), we measured an average eviction rate of 99.93% and an average execution time of 4 275 CPU cycles. We observed that LRU eviction ($A = 1$, $D = 1$) on the ARM Cortex-

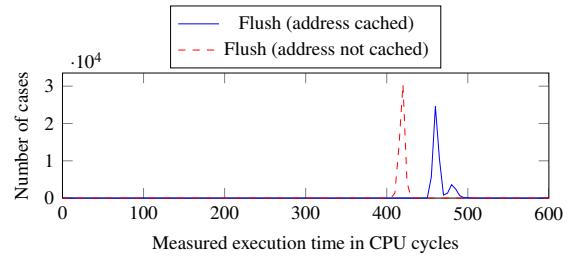


Figure 3: Histograms of the execution time of the flush operation on cached and not cached addresses measured on the Samsung Galaxy S6.

A53 would take 28 times more CPU cycles to achieve an average eviction rate of only 99.10%, thus it is not suitable for attacks on the last-level cache as used in previous work [51]. The reason for this is that data can only be allocated to L2 cache by evicting it from the L1 cache on the ARM Cortex-A53. Therefore, it is better to reaccess the data that is already in the L2 cache and gradually add new addresses to the set of cached addresses instead of accessing more different addresses.

On the ARM Cortex-A57 the userspace flush instruction was significantly faster in any case. Thus, for *Flush+Reload* we use the flush instruction and for *Prime+Probe* the eviction strategy. Falling back to *Evict+Reload* is not necessary on the Cortex-A57. Similarly to recent Intel x86 CPUs, the execution time of the flush instruction on ARM depends on whether or not the value is cached, as shown in Figure 3. The execution time is higher if the address is not cached and lower if the address is cached. This observation allows us to distinguish between cache hits and cache misses depending on the timing behavior of the flush instruction, and therefore to perform a *Flush+Flush* attack. Thus, in case of shared memory between the victim and the attacker, it is not even required to evict and reload an address in order to exploit the cache side channel.

A note on Prime+Probe. Finding a fast eviction strategy for *Prime+Probe* on architectures with a random replacement policy is not as straightforward as on Intel x86. Even in case of x86 platforms, the problem of cache trashing has been discussed by Tromer et al. [54]. Cache trashing occurs when reloading (probing) an address evicts one of the addresses that are to be accessed next. While Tromer et al. were able to overcome this problem by using a doubly-linked list that is accessed forward during the prime step and backwards during the probe step, the random replacement policy on ARM also contributes to the negative effect of cache trashing.

We analyzed the behavior of the cache and designed a prime step and a probe step that work with a smaller set size to avoid set thrashing. Thus, we set the eviction set size to 15 on the Alcatel One Touch Pop 2. As we run the *Prime+Probe* attack in a loop, exactly 1 way in the L2 cache will not be occupied after a few attack rounds. We might miss a victim access in $\frac{1}{16}$ of the cases, which however is necessary as otherwise we would not be able to get reproducible measurements at all due to set thrashing. If the victim replaces one of the 15 ways occupied by the attacker, there is still one free way to reload the address that was evicted. This reduces the chance of set thrashing significantly and allows us to successfully perform *Prime+Probe* on caches with a random replacement policy.

3.3 Accurate Unprivileged Timing

In this section, we tackle the aforementioned challenge 5, *i.e.*, cycle-accurate timings require root access on ARM.

In order to distinguish cache hits and cache misses, timing sources or dedicated performance counters can be used. We focus on timing sources, as cache misses have a significantly higher access latency and timing sources are well studied on Intel x86 CPUs. Cache attacks on x86 CPUs employ the unprivileged `rdtsc` instruction to obtain a sub-nanosecond resolution timestamp. The ARMv7-A architecture does not provide an instruction for this purpose. Instead, the ARMv7-A architecture has a performance monitoring unit that allows to monitor CPU activity. One of these performance counters—denoted as *cycle count register* (PMCCNTR)—can be used to distinguish cache hits and cache misses by relying on the number of CPU cycles that passed during a memory access. However, these performance counters are not accessible from userspace by default and an attacker would need root privileges.

We broaden the attack surface by exploiting timing sources that are accessible without any privileges or permissions. We identified three possible alternatives for timing measurements.

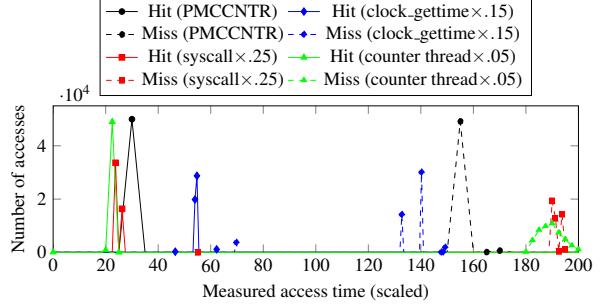


Figure 4: Histogram of cross-core cache hits/misses on the Alcatel One Touch Pop 2 using different methods. X-values are scaled for visual representation.

Unprivileged syscall. The `perf_event_open` syscall is an abstract layer to access performance information through the kernel independently of the underlying hardware. For instance, `PERF_COUNT_HW_CPU_CYCLES` returns an accurate cycle count including a minor overhead due to the syscall. The availability of this feature depends on the Android kernel configuration, *e.g.*, the stock kernel on the Alcatel One Touch Pop 2 as well as the OnePlus One provide this feature by default. Thus, in contrast to previous work [51], the attacker does not have to load a kernel module to access this information as the `perf_event_open` syscall can be accessed without any privileges or permissions.

POSIX function. Another alternative to obtain sufficiently accurate timing information is the POSIX function `clock_gettime()`, with an accuracy in the range of microseconds to nanoseconds. Similar information can also be obtained from `/proc/timer_list`.

Dedicated thread timer. If no interface with sufficient accuracy is available, an attacker can run a thread that increments a global variable in a loop, providing a fair approximation of a cycle counter. Our experiments show that this approach works reliably on smartphones as well as recent x86 CPUs. The resolution of this threaded timing information is as high as with the other methods.

In Figure 4 we show the cache hit and miss histogram based on the four different methods, including the cycle count register, on a Alcatel One Touch Pop 2. Despite the latency and noise, cache hits and cache misses are clearly distinguishable with all approaches. Thus, all methods can be used to implement cache attacks. Determining the best timing method on the device under attack can be done in a few seconds during an online attack.

4 High Performance Covert Channels

To evaluate the performance of our attacks, we measure the capacity of cross-core and cross-CPU cache covert channels. A covert channel enables two unprivileged applications on a system to communicate with each other without using any data transfer mechanisms provided by the operating system. This communication evades the sandboxing concept and the permission system (cf. collusion attacks [36]). Both applications were running in the background while the phone was mostly idle and an unrelated app was running as the foreground application.

Our covert channel is established on addresses of a shared library that is used by both the sender and the receiver. While both processes have read-only access to the shared library, they can transmit information by loading addresses from the shared library into the cache or evicting (flushing) it from the cache, respectively.

The covert channel transmits packets of n -bit data, an s -bit sequence number, and a c -bit checksum that is computed over data and sequence number. The sequence number is used to distinguish consecutive packets and the checksum is used to check the integrity of the packet. The receiver acknowledges valid packets by responding with an s -bit sequence number and an x -bit checksum. By adjusting the sizes of checksums and sequence numbers the error rate of the covert channel can be controlled.

Each bit is represented by one address in the shared library, whereas no two addresses are chosen that map to the same cache set. To transmit a bit value of 1, the sender accesses the corresponding address in the library. To transmit a bit value of 0, the sender does not access the corresponding address, resulting in a cache miss on the receiver's side. Thus, the receiving process observes a cache hit or a cache miss depending on the memory access performed by the sender. The same method is used for the acknowledgements sent by the receiving process.

We implemented this covert channel using *Evict+Reload*, *Flush+Reload*, and *Flush+Flush* on our smartphones. The results are summarized in Table 4. On the Samsung Galaxy S6, we achieve a cross-core transmission rate of 1 140 650 bps at an error rate of 1.10%. This is 265 times faster than any existing covert channel on smartphones. In a cross-CPU transmission we achieve a transmission rate of 257 509 bps at an error rate of 1.83%. We achieve a cross-core transition rate of 178 292 bps at an error rate of 0.48% using *Flush+Flush* on the Samsung Galaxy S6. On the Alcatel One Touch Pop 2 we achieve a cross-core transmission rate of 13 618 bps at an error rate of 3.79% using *Evict+Reload*. This is still 3 times faster than previous covert channels on smartphones. The covert channel is significantly slower on the Alcatel One Touch Pop 2 than on the Samsung Galaxy S6 because the hardware is much

slower, *Evict+Reload* is slower than *Flush+Reload*, and retransmission might be necessary in 0.14% of the cases where eviction is not successful (cf. Section 3.2). On the older OnePlus One we achieve a cross-core transmission rate of 12 537 bps at an error rate of 5.00%, 3 times faster than previous covert channels on smartphones. The reason for the higher error rate is the additional timing noise due to the cache coherence protocol performing a high number of remote-core fetches.

5 Attacking User Input on Smartphones

In this section we demonstrate cache side-channel attacks on Android smartphones. We implement cache template attacks [19] to create and exploit accurate cache-usage profiles using the *Evict+Reload* or *Flush+Reload* attack. Cache template attacks have a profiling phase and an exploitation phase. In the profiling phase, a template matrix is computed that represents how many cache hits occur on a specific address when triggering a specific event. The exploitation phase uses this matrix to infer events from cache hits.

To perform cache template attacks, an attacker has to map shared binaries or shared libraries as read-only shared memory into its own address space. By using shared libraries, the attacker bypasses any potential countermeasures taken by the operating system, such as restricted access to runtime data of other apps or address space layout randomization (ASLR). The attack can even be performed online on the device under attack if the event can be simulated.

Triggering the actual event that an attacker wants to spy on might require either (1) an offline phase or (2) privileged access. For instance, in case of a keylogger, the attacker can gather a cache template matrix offline for a specific version of a library, or the attacker relies on privileged access of the application (or a dedicated permission) in order to be able to simulate events for gathering the cache template matrix. However, the actual exploitation of the cache template matrix to infer events neither requires privileged access nor any permission.

5.1 Attacking a Shared Library

Just as Linux, Android uses a large number of shared libraries, each with a size of up to several megabytes. We inspected all available libraries on the system by manually scanning the names and identified libraries that might be responsible for handling user input, e.g., the `libinput.so` library. Without loss of generality, we restricted the set of attacked libraries since testing all libraries would have taken a significant amount of time. Yet, an adversary could exhaustively probe all libraries.

Table 4: Comparison of covert channels on Android.

Work	Type	Bandwidth [bps]	Error rate
Ours (Samsung Galaxy S6)	<i>Flush+Reload</i> , cross-core	1 140 650	1.10%
Ours (Samsung Galaxy S6)	<i>Flush+Reload</i> , cross-CPU	257 509	1.83%
Ours (Samsung Galaxy S6)	<i>Flush+Flush</i> , cross-core	178 292	0.48%
Ours (Alcatel One Touch Pop 2)	<i>Evict+Reload</i> , cross-core	13 618	3.79%
Ours (OnePlus One)	<i>Evict+Reload</i> , cross-core	12 537	5.00%
Marforio et al. [36]	Type of Intents	4 300	—
Marforio et al. [36]	UNIX socket discovery	2 600	—
Schlegel et al. [48]	File locks	685	—
Schlegel et al. [48]	Volume settings	150	—
Schlegel et al. [48]	Vibration settings	87	—

We automated the search for addresses in these shared libraries and after identifying addresses, we monitored them in order to infer user input events. For instance, in the profiling phase on `libinput.so`, we simulated events via the android-debug bridge (adb shell) with two different methods. The first method uses the `input` command line tool to simulate user input events. The second method is writing event messages to `/dev/input/event*`. Both methods can run entirely on the device for instance in idle periods while the user is not actively using the device. As the second method only requires a `write()` statement it is significantly faster, but it is also more device specific. Therefore, we used the `input` command line except when profiling differences between different letter keys. While simulating these events, we simultaneously probed all addresses within the `libinput.so` library, *i.e.*, we measured the number of cache hits that occurred on each address when triggering a specific event. As already mentioned above, the simulation of some events might require either an offline phase or specific privileges in case of online attacks.

Figure 5 shows part of the cache template matrix for `libinput.so`. We triggered the following events: key events including the power button (`key`), long touch events (`longpress`), `swipe` events, touch events (`tap`), and text input events (`text`) via the `input` tool as often as possible and measured each address and event for one second. The cache template matrix clearly reveals addresses with high cache-hit rates for specific events. Darker colors represent addresses with higher cache-hit rates for a specific event and lighter colors represent addresses with lower cache-hit rates. Hence, we can distinguish different events based on cache hits on these addresses.

We verified our results by monitoring the identified addresses while operating the smartphone manually, *i.e.*, we touched the screen and our attack application reliably reported cache hits on the monitored addresses. For instance, address `0x11040` of `libinput.so` can be used to distinguish tap actions and swipe actions on the screen of the Alcatel One Touch Pop 2. Tap actions cause a smaller

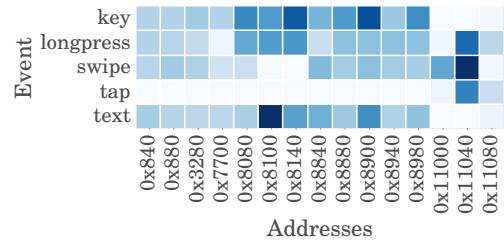


Figure 5: Cache template matrix for `libinput.so`.

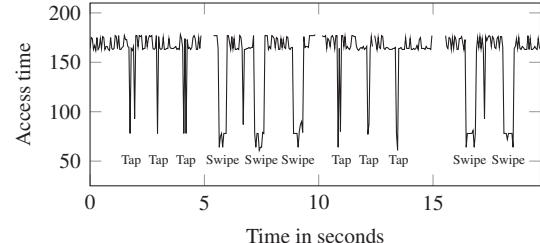


Figure 6: Monitoring address `0x11040` of `libinput.so` on the Alcatel One Touch Pop 2 reveals taps and swipes.

number of cache hits than swipe actions. Swipe actions cause cache hits in a high frequency as long as the screen is touched. Figure 6 shows a sequence of 3 tap events, 3 swipe events, 3 tap events, and 2 swipe events. These events can be clearly distinguished due to the fast access times. The gaps mark periods of time where our program was not scheduled on the CPU. Events occurring in those periods can be missed by our attack.

Swipe input allows to enter words by swiping over the soft-keyboard and thereby connecting single characters to form a word. Since we are able to determine the length of swipe movements, we can correlate the length of the swipe movement with the actual word length in any Android application or system interface that uses swipe input without any privileges. Furthermore, we can determine the actual length of the unlock pattern for the pattern-unlock mechanism.

Figure 7 shows a user input sequence consisting of 3 tap events and 3 swipe events on the Samsung Galaxy S6. The attack was conducted using *Flush+Reload*. An attacker can monitor every single event. Taps and swipes can be distinguished based on the length of the cache hit phase. The length of a swipe movement can be determined from the same information. Figure 8 shows the same experiment on the OnePlus One using *Evict+Reload*. Thus, our attack techniques work on coherent non-inclusive last-level caches.

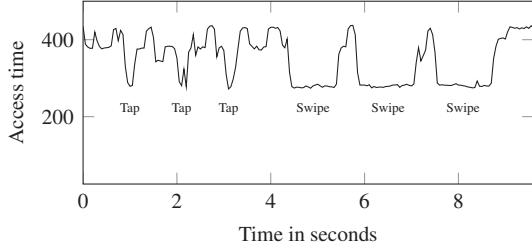


Figure 7: Monitoring address 0xDC5C of libinput.so on the Samsung Galaxy S6 reveals tap and swipe events.

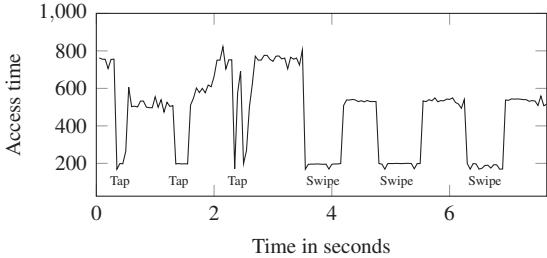


Figure 8: Monitoring address 0xBFF4 of libinput.so on the OnePlus One reveals tap and swipe events.

5.2 Attacking ART Binaries

Instead of attacking shared libraries, it is also possible to apply this attack to ART (Android Runtime) executables [1] that are compiled ahead of time. We used this attack on the default AOSP keyboard and evaluated the number of accesses to every address in the optimized executable that responds to an input of a letter on the keyboard. It is possible to find addresses that correspond to a key press and more importantly to distinguish between taps and key presses. Figure 9 shows the corresponding cache template matrix. We summarize the letter keys in one line (*alphabet*) as they did not vary significantly. These addresses can be used to monitor key presses on the keyboard. We identified an address that corresponds only to letters on the keyboard and hardly on the space bar or the return button. With this information it is pos-

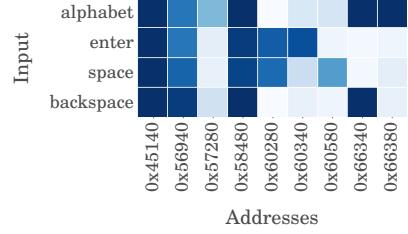


Figure 9: Cache template matrix for the default AOSP keyboard.

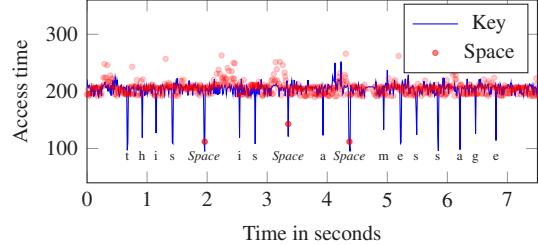


Figure 10: *Evict+Reload* on 2 addresses in custpack@ app@withoutlibs@LatinIME.apk@classes.dex on the Alcatel One Touch Pop 2 while entering the sentence “this is a message”.

sible to precisely determine the length of single words entered using the default AOSP keyboard.

We illustrate the capability of detecting word lengths in Figure 10. The blue line shows the timing measurements for the address identified for keys in general, the red dots represent measurements of the address for the space key. The plot shows that we can clearly determine the length of entered words and monitor user input accurately over time.

5.3 Discussion and Impact

Our proof-of-concept attacks exploit shared libraries and binaries from Android apk files to infer key strokes. The cache template attack technique we used for these attacks is generic and can also be used to attack any other library. For instance, there are various libraries that handle different hardware modules and software events on the device, such as GPS, Bluetooth, camera, NFC, vibrator, audio and video decoding, web and PDF viewers. Each of these libraries contains code that is executed and data that is accessed when the device is in use. Thus, an attacker can perform a cache template attack on any of these libraries and spy on the corresponding device events. For instance, our attack can be used to monitor activity of the GPS sensor, bluetooth, or the camera. An attacker can record such user activities over time to learn more about the user.

We can establish inter-keystroke timings at an accuracy as high as the accuracy of cache side-channel attacks on keystrokes on x86 systems with a physical keyboard. Thus, the inter-keystroke timings can be used to infer entered words, as has been shown by Zhang et al. [61]. Our attack even has a higher resolution than [61], *i.e.*, it is sub-microsecond accurate. Furthermore, we can distinguish between keystrokes on the soft-keyboard and generic touch actions outside the soft-keyboard. This information can be used to enhance sensor-based keyloggers that infer user input on mobile devices by exploiting, *e.g.*, the accelerometer and the gyroscope [7, 11, 12, 39, 59] or the ambient-light sensor [49]. However, these attacks suffer from a lack of knowledge when exactly a user touches the screen. Based on our attack, these sensor-based keyloggers can be improved as our attack allows to infer (1) the exact time when the user touches the screen, and (2) whether the user touches the soft-keyboard or any other region of the display.

Our attacks only require the user to install a malicious app on the smartphone. However, as shown by Oren et al. [42], *Prime+Probe* attacks can even be performed from within browser sandboxes through remote websites using JavaScript on Intel platforms. Gruss et al. [16] showed that JavaScript timing measurements in web browsers on ARM-based smartphones achieve a comparable accuracy as on Intel platforms. Thus, it seems likely that *Prime+Probe* through a website works on ARM-based smartphones as well. We expect that such attacks will be demonstrated in future work. The possibility of attacking millions of users shifts the focus of cache attacks to a new range of potential malicious applications.

In our experiments with the predecessor of ART, the Dalvik VM, we found that the just-in-time compilation effectively prevents *Evict+Reload* and *Flush+Reload* attacks. The just-in-time compiled code is not shared and thus the requirements for these two attacks are not met. However, *Prime+Probe* attacks work on ART binaries and just-in-time compiled Dalvik VM code likewise.

6 Attack on Cryptographic Algorithms

In this section we show how *Flush+Reload*, *Evict+Reload*, and *Prime+Probe* can be used to attack AES T-table implementations that are still in use on Android devices. Furthermore, we demonstrate the possibility to infer activities within the ARM TrustZone by observing the cache activity using *Prime+Probe*. We perform all attacks cross-core and in a synchronized setting, *i.e.*, the attacker triggers the execution of cryptographic algorithms by the victim process. Although more sophisticated attacks are possible, our goal is to demonstrate that our work enables practical cache attacks on smartphones.

6.1 AES T-Table Attacks

Many cache attacks against AES T-table implementations have been demonstrated and appropriate countermeasures have already been proposed. Among these countermeasures are, *e.g.*, so-called bit-sliced implementations [29, 32, 46]. Furthermore, Intel addressed the problem by adding dedicated instructions for AES [20] and ARM also follows the same direction with the ARMv8 instruction set [4]. However, our investigations showed that Bouncy Castle, a crypto library widely used in Android apps such as the WhatsApp messenger [2], still uses a T-table implementation. Moreover, the OpenSSL library, which is the default crypto provider on recent Android versions, uses T-table implementations until version 1.0.1.³ This version is still officially supported and commonly used on Android devices, *e.g.*, the Alcatel One Touch Pop 2. T-tables contain the pre-computed AES round transformations, allowing to perform encryptions and decryptions by simple XOR operations. For instance, let p_i denote the plaintext bytes, k_i the initial key bytes, and $s_i = p_i \oplus k_i$ the initial state bytes. The initial state bytes are used to retrieve pre-computed T-table elements for the next round. If an attacker knows a plaintext byte p_i and the accessed element of the T-table, it is possible to recover the key bytes $k_i = s_i \oplus p_i$. However, it is only possible to derive the upper 4 bits of k_i through our cache attack on a device with a cache line size of 64 bytes. This way, the attacker can learn 64 key bits. In second-round and last-round attacks the key space can be reduced further. For details about the basic attack strategy we refer to the work of Osvik et al. [43, 54]. Although we successfully mounted an *Evict+Reload* attack on the Alcatel One Touch Pop 2 against the OpenSSL AES implementation, we do not provide further insights as we are more interested to perform the first cache attack on a Java implementation.

Attack on Bouncy Castle. Bouncy Castle is implemented in Java and provides various cryptographic primitives including AES. As Bouncy Castle 1.5 still employs AES T-table implementations by default, all Android devices that use this version are vulnerable to our presented attack. To the best of our knowledge, we are the first to show an attack on a Java implementation.

During the initialization of Bouncy Castle, the T-tables are copied to a local private memory area. Therefore, these copies are not shared among different processes. Nevertheless, we demonstrate that *Flush+Reload* and *Evict+Reload* are efficient attacks on such an implemen-

³Later versions use a bit-sliced implementation if ARM NEON is available or dedicated AES instructions if ARMv8-A instructions are available. Otherwise, a T-table implementation is used. This is also the case for Google’s BoringSSL library.

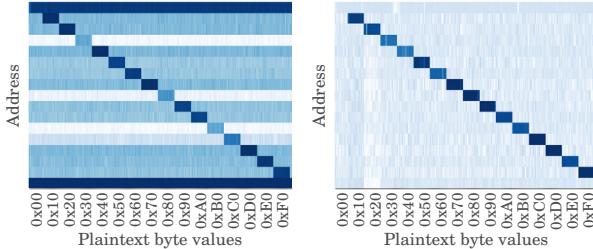


Figure 11: Attack on Bouncy Castle’s AES using *Evict+Reload* on the Alcatel One Touch Pop 2 (left) and *Flush+Reload* on the Samsung Galaxy S6 (right).

tation if shared memory is available. Further, we demonstrate a cross-core *Prime+Probe* attack without shared memory that is applicable in a real-world scenario.

Figure 11 shows a template matrix of the first T-table for all 256 values for plaintext byte p_0 and a key that is fixed to 0 while the remaining plaintext bytes are random. These plots reveal the upper 4 key bits of k_0 [43, 51]. Thus, in our case the key space is reduced to 64 bits after 256–512 encryptions. We consider a first-round attack only, because we aim to demonstrate the applicability of these attacks on ARM-based mobile devices. However, full-key recovery is possible with the same techniques by considering more sophisticated attacks targeting different rounds [47, 54], even for asynchronous attackers [22, 26].

We can exploit the fact that the T-tables are placed on a different boundary every time the process is started. By restarting the victim application we can obtain arbitrary disalignments of T-tables. Disaligned T-tables allow to reduce the key space to 20 bits on average and for specific disalignments even full-key recovery without a single brute-force computation is possible [51, 53]. We observed not a single case where the T-tables were aligned. Based on the first-round attack matrix in Figure 11, the expected number of encryptions until a key byte is identified is $1.81 \cdot 128$. Thus, full key recovery is possible after $1.81 \cdot 128 \cdot 16 = 3707$ encryptions by monitoring a single address during each encryption.

Real-world cross-core attack on Bouncy Castle. If the attacker has no way to share a targeted memory region with the victim, *Prime+Probe* instead of *Evict+Reload* or *Flush+Reload* can be used. This is the case for dynamically generated data or private memory of another process. Figure 12 shows the *Prime+Probe* histogram for cache hits and cache misses. We observe a higher execution time if the victim accesses a congruent memory location. Thus, *Prime+Probe* can be used for a real-world cross-core attack on Bouncy Castle and also allows to exploit disaligned T-tables as mentioned above.

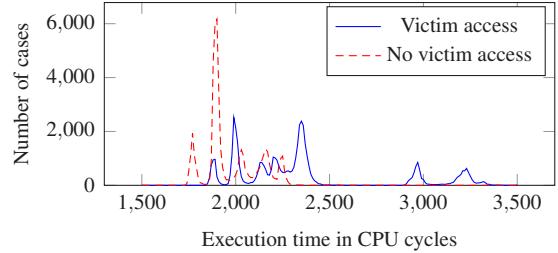


Figure 12: Histogram of *Prime+Probe* timings depending on whether the victim accesses congruent memory on the ARM Cortex-A53.

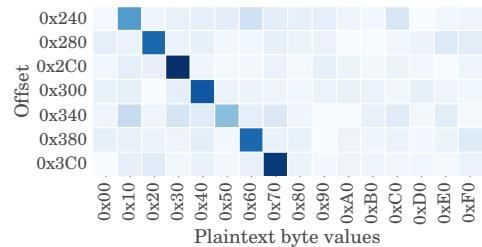


Figure 13: Excerpt of the attack on Bouncy Castle’s AES using *Prime+Probe*.

In a preprocessing step, the attacker identifies the cache sets to be attacked by performing random encryptions and searching for active cache sets. Recall that the cache set (index) is derived directly from the physical address on ARM, *i.e.*, the lowest n bits determine the offset within a 2^n -byte cache line and the next s bits determine one of the 2^s cache sets. Thus, we only have to find a few cache sets where a T-table maps to in order to identify all cache sets required for the attack. On x86 the replacement policy facilitates this attack and allows even to deduce the number of ways that have been replaced in a specific cache set [43]. On ARM the random replacement policy makes *Prime+Probe* more difficult as cache lines are replaced in a less predictable way. To launch a *Prime+Probe* attack, we apply the eviction strategy and the crafted reaccess patterns we described in Section 3.2.

Figure 13 shows an excerpt of the cache template matrix resulting from a *Prime+Probe* attack on one T-table. For each combination of plaintext byte and offset we performed 100 000 encryptions for illustration purposes. We only need to monitor a single address to obtain the upper 4 bits of s_i and, thus, the upper 4 bits of $k_i = s_i \oplus p_i$. Compared to the *Evict+Reload* attack from the previous section, *Prime+Probe* requires 3 times as many measurements to achieve the same accuracy. Nevertheless, our results show that an attacker can run *Prime+Probe* attacks on ARM CPUs just as on Intel CPUs.

6.2 Spy on TrustZone Code Execution

The ARM TrustZone is a hardware-based security technology built into ARM CPUs to provide a secure execution environment [4]. This trusted execution environment is isolated from the *normal world* using hardware support. The TrustZone is used, e.g., as a hardware-backed credential store, to emulate secure elements for payment applications, digital rights management as well as verified boot and kernel integrity measurements. The services are provided by so-called trustlets, *i.e.*, applications that run in the secure world.

Since the secure monitor can only be called from the supervisor context, the kernel provides an interface for the userspace to interact with the TrustZone. On the Alcatel One Touch Pop 2, the TrustZone is accessible through a device driver called QSEECom (Qualcomm Secure Execution Environment Communication) and a library `libQSEEComAPI.so`. The key master trustlet on the Alcatel One Touch Pop 2 provides an interface to generate hardware-backed RSA keys, which can then be used inside the TrustZone to sign and verify signatures.

Our observations showed that a *Prime+Probe* attack on the TrustZone is not much different from a *Prime+Probe* attack on any application in the normal world. However, as we do not have access to the source code of the TrustZone OS or any trustlet, we only conduct simple attacks.⁴ We show that *Prime+Probe* can be used to distinguish whether a provided key is valid or not. While this might also be observable through the overall execution time, we demonstrate that the TrustZone isolation does not protect against cache attacks from the normal world and any trustlet can be attacked.

We evaluated cache profiles for multiple valid as well as invalid keys. Figure 14 shows the mean squared error over two runs for different valid keys and one invalid key compared to the average of valid keys. We performed *Prime+Probe* before and after the invocation of the corresponding trustlet, *i.e.*, prime before the invocation and probe afterwards. We clearly see a difference in some sets (cache sets 250–320) that are used during the signature generation using a valid key. These cache profiles are reproducible and can be used to distinguish whether a valid or an invalid key has been used in the TrustZone. Thus, the secure world leaks information to the non-secure world.

On the Samsung Galaxy S6, the TrustZone flushes the cache when entering or leaving the trusted world. However, by performing a *Prime+Probe* attack in parallel, *i.e.*, multiple times while the trustlet performs the corresponding computations, the same attack can be mounted.

⁴More sophisticated attacks would be possible by reverse engineering these trustlets.

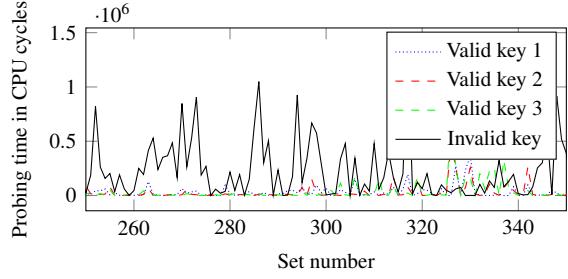


Figure 14: Mean squared error between the average *Prime+Probe* timings of valid keys and invalid keys on the Alcatel One Touch Pop 2.

7 Countermeasures

Although our attacks exploit hardware weaknesses, software-based countermeasures could impede such attacks. Indeed, we use unprotected access to system information that is available on all Android versions.

As we have shown, the operating system cannot prevent access to timing information. However, other information supplied by the operating system that facilitates these attacks could be restricted. For instance, we use `/proc/pid/` to retrieve information about any other process on the device, *e.g.*, `/proc/pid/pagemap` is used to resolve virtual addresses to physical addresses. Even though access to `/proc/pid/pagemap` and `/proc/self/pagemap` has been restricted in Linux in early 2015, the Android kernel still allows access to these resources. Given the immediately applicable attacks we presented, we stress the urgency to merge the corresponding patches into the Android kernel. Furthermore, we use `/proc/pid/maps` to determine shared objects that are mapped into the address space of a victim. Restricting access to `procfs` to specific privileges or permissions would make attacks harder. We recommend this for both the Linux kernel as well as Android.

We also exploit the fact that access to shared libraries as well as `dex` and `art` optimized program binaries is only partially restricted on the file system level. While we cannot retrieve a directory listing of `/data/dalvik-cache/`, all files are readable for any process or Android application. We recommend to allow read access to these files to their respective owner exclusively to prevent *Evict+Reload*, *Flush+Reload*, and *Flush+Flush* attacks through these shared files.

In order to prevent cache attacks against AES T-tables, hardware instructions should be used. If this is not an option, a software-only bit-sliced implementation must be employed, especially when disalignment is possible, as it is the case in Java. Since OpenSSL 1.0.2 a bit-sliced implementation is available for devices capable of the ARM

NEON instruction set and dedicated AES instructions are used on ARMv8-A devices. Cryptographic algorithms can also be protected using cache partitioning [34]. However, cache partitioning comes with a performance impact and it can not prevent all attacks, as the number of cache partitions is limited.

We responsibly disclosed our attacks and the proposed countermeasures to Google and other development groups prior to the publication of our attacks. Google has applied upstream patches preventing access to `/proc/pid/pagemap` in early 2016 and recommended installing the security update in March 2016 [15].

8 Conclusion

In this work we demonstrated the most powerful cross-core cache attacks *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, and *Flush+Flush* on default configured unmodified Android smartphones. Furthermore, these attacks do not require any permission or privileges. In order to enable these attacks in real-world scenarios, we have systematically solved all challenges that prevented highly accurate cache attacks on ARM so far. Our attacks are the first cross-core and cross-CPU attacks on ARM CPUs. Furthermore, our attack techniques provide a high resolution and a high accuracy, which allows monitoring singular events such as touch and swipe actions on the screen, touch actions on the soft-keyboard, and inter-keystroke timings. In addition, we show that efficient state-of-the-art key-recovery attacks can be mounted against the default AES implementation that is part of the Java Bouncy Castle crypto provider and that cache activity in the ARM TrustZone can be monitored from the normal world.

The presented example attacks are by no means exhaustive and launching our proposed attack against other libraries and apps will reveal numerous further exploitable information leaks. Our attacks are applicable to hundreds of millions of today’s off-the-shelf smartphones as they all have very similar if not identical hardware. This is especially daunting since smartphones have become the most important personal computing devices and our techniques significantly broaden the scope and impact of cache attacks.

Acknowledgment

We would like to thank our anonymous reviewers for their valuable comments and suggestions.



Supported by the EU Horizon 2020 programme under GA No. 644052 (HECTOR), the EU FP7 programme under GA No. 610436 (MATTHEW), and the Austrian Research Promotion Agency (FFG) under grant number 845579 (MEMSEC).

References

- [1] ANDROID OPEN SOURCE PROJECT. Configuring ART. <https://source.android.com/devices/tech/dalvik/configure.html>, Nov. 2015. Retrieved on November 10, 2015.
- [2] APPTORNADO. AppBrain - Android library statistics - Spongy Castle - Bouncy Castle for Android. <http://www.appbrain.com/stats/libraries/details/spongycastle/spongycastle-bouncy-castle-for-android>, June 2016. Retrieved on June 6, 2016.
- [3] ARM LIMITED. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. ARM Limited, 2012.
- [4] ARM LIMITED. *ARM Architecture Reference Manual ARMv8*. ARM Limited, 2013.
- [5] ARM LIMITED. *ARM Cortex-A57 MPCore Processor Technical Reference Manual r1p0*. ARM Limited, 2013.
- [6] ARM LIMITED. *ARM Cortex-A53 MPCore Processor Technical Reference Manual r0p3*. ARM Limited, 2014.
- [7] AVIV, A. J., SAPP, B., BLAZE, M., AND SMITH, J. M. Practicality of Accelerometer Side Channels on Smartphones. In *Annual Computer Security Applications Conference – ACSAC* (2012), ACM, pp. 41–50.
- [8] BENGERT, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. “Ooh Aah... Just a Little Bit”: A Small Amount of Side Channel Can Go a Long Way. In *Cryptographic Hardware and Embedded Systems – CHES* (2014), vol. 8731 of *LNCS*, Springer, pp. 75–92.
- [9] BERNSTEIN, D. J. Cache-Timing Attacks on AES, 2004. URL: <http://cr.yp.to/papers.html#cachetiming>.
- [10] BOGDANOV, A., EISENBARTH, T., PAAR, C., AND WIENECKE, M. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In *Topics in Cryptology – CT-RSA* (2010), vol. 5985 of *LNCS*, Springer, pp. 235–251.
- [11] CAI, L., AND CHEN, H. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *USENIX Workshop on Hot Topics in Security – HotSec* (2011), USENIX Association.
- [12] CAI, L., AND CHEN, H. On the Practicality of Motion Based Keystroke Inference Attack. In *Trust and Trustworthy Computing – TRUST* (2012), vol. 7344 of *LNCS*, Springer, pp. 273–290.
- [13] GALLAIS, J., KIZHVATOV, I., AND TUNSTALL, M. Improved Trace-Driven Cache-Collision Attacks against Embedded AES Implementations. In *Workshop on Information Security Applications – WISA* (2010), vol. 6513 of *LNCS*, Springer, pp. 243–257.
- [14] GALLAIS, J.-F., AND KIZHVATOV, I. Error-Tolerance in Trace-Driven Cache Collision Attacks. In *COSADE* (2011), pp. 222–232.
- [15] GOOGLE INC. Nexus Security Bulletin - March 2016. <https://source.android.com/security/bulletin/2016-03-01.html>, Mar. 2016. Retrieved on June 6, 2016.
- [16] GRUSS, D., BIDNER, D., AND MANGARD, S. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *European Symposium on Research – ESORICS* (2015), vol. 9326 of *LNCS*, Springer, pp. 108–122.

- [17] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA'16* (2016).
- [18] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA'16* (2016).
- [19] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015), USENIX Association, pp. 897–912.
- [20] GUERON, S. White Paper: Intel Advanced Encryption Standard (AES) Instructions Set, 2010. URL: <https://software.intel.com/file/24917>.
- [21] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy – S&P* (2011), IEEE Computer Society, pp. 490–505.
- [22] GÜLMEZOGLU, B., INCI, M. S., APECECHEA, G. I., EISENBARTH, T., AND SUNAR, B. A Faster and More Realistic Flush+Reload Attack on AES. In *Constructive Side-Channel Analysis and Secure Design – COSADE* (2015), vol. 9064 of *LNCS*, Springer, pp. 111–126.
- [23] HUND, R., WILLEMS, C., AND HOLZ, T. Practical Timing Side-Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy – S&P* (2013), IEEE, pp. 191–205.
- [24] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. SSA: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *IEEE Symposium on Security and Privacy – S&P* (2015), IEEE Computer Society.
- [25] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cross Processor Cache Attacks. In *ACM Computer and Communications Security – ASIACCS* (2016), ACM, pp. 353–364.
- [26] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses Symposium – RAID* (2014), vol. 8688 of *LNCS*, Springer, pp. 299–319.
- [27] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Know Thy Neighbor: Crypto Library Detection in Cloud. *Privacy Enhancing Technologies 1*, 1 (2015), 25–40.
- [28] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Lucky 13 Strikes Back. In *ACM Computer and Communications Security – ASIACCS* (2015), ACM, pp. 85–96.
- [29] KÄSPER, E., AND SCHWABE, P. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems – CHES* (2009), vol. 5747 of *LNCS*, Springer, pp. 1–17.
- [30] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security* 8, 2/3 (2000), 141–158.
- [31] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO* (1996), vol. 1109 of *LNCS*, Springer, pp. 104–113.
- [32] KÖNIGHOFER, R. A Fast and Cache-Timing Resistant Implementation of the AES. In *Topics in Cryptology – CT-RSA* (2008), vol. 4964 of *LNCS*, Springer, pp. 187–202.
- [33] LAL SHIMPI, ANANDTECH. Answered by the Experts: ARM's Cortex A53 Lead Architect, Peter Greenhalgh. <http://www.anandtech.com/show/7591/answered-by-the-experts-arm-s-cortex-a53-lead-architect-peter-greenhalgh>, Dec. 2013. Retrieved on November 10, 2015.
- [34] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C. V., HEISER, G., AND LEE, R. B. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture – HPCA* (2016), IEEE Computer Society, pp. 406–418.
- [35] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy – SP* (2015), IEEE Computer Society, pp. 605–622.
- [36] MARFORIO, C., RITZDORF, H., FRANCILLON, A., AND CAPKUN, S. Analysis of the Communication Between Colluding Applications on Modern Smartphones. In *Annual Computer Security Applications Conference – ACSAC* (2012), ACM, pp. 51–60.
- [37] MAURICE, C., NEUMANN, C., HEEN, O., AND FRANCILLON, A. C5: Cross-Cores Cache Covert Channel. In *Detection of Intrusions and Malware, and Vulnerability Assessment – DIMVA* (2015), vol. 9148 of *LNCS*, Springer, pp. 46–64.
- [38] MAURICE, C., SCOURNEC, N. L., NEUMANN, C., HEEN, O., AND FRANCILLON, A. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses – RAID* (2015), vol. 9404 of *LNCS*, Springer, pp. 48–65.
- [39] MILUZZO, E., VARSHAVSKY, A., BALAKRISHNAN, S., AND CHOUDHURY, R. R. Tapprints: Your Finger Taps Have Fingerprints. In *Mobile Systems, Applications, and Services – MobiSys* (2012), ACM, pp. 323–336.
- [40] NEVE, M. *Cache-based Vulnerabilities and SPAM Analysis*. PhD thesis, UCL, 2006.
- [41] NEVE, M., SEIFERT, J., AND WANG, Z. A Refined Look at Bernstein's AES Side-Channel Analysis. In *ACM Computer and Communications Security – ASIACCS* (2006), ACM, p. 369.
- [42] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Conference on Computer and Communications Security – CCS* (2015), ACM, pp. 1406–1418.
- [43] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA* (2006), vol. 3860 of *LNCS*, Springer, pp. 1–20.
- [44] PAGE, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptology ePrint Archive 2002/169*.
- [45] PERCIVAL, C. Cache Missing for Fun and Profit, 2005. URL: <http://daemonology.net/hyperthreading-considered-harmful/>.
- [46] REBEIRO, C., SELVAKUMAR, A. D., AND DEVI, A. S. L. Bit-slice Implementation of AES. In *Cryptology and Network Security – CANS* (2006), vol. 4301 of *LNCS*, Springer, pp. 203–212.
- [47] SAVAS, E., AND YILMAZ, C. A Generic Method for the Analysis of a Class of Cache Attacks: A Case Study for AES. *Comput. J.* 58, 10 (2015), 2716–2737.
- [48] SCHLEGEL, R., ZHANG, K., ZHOU, X., INTWALA, M., KAPADIA, A., AND WANG, X. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Network and Distributed System Security Symposium – NDSS* (2011), The Internet Society.
- [49] SPREITZER, R. PIN Skimming: Exploiting the Ambient-Light Sensor in Mobile Devices. In *Security and Privacy in Smartphones & Mobile Devices – SPSM@CCS* (2014), ACM, pp. 51–62.
- [50] SPREITZER, R., AND GÉRARD, B. Towards More Practical Time-Driven Cache Attacks. In *Information Security Theory and Practice – WISTP* (2014), vol. 8501 of *LNCS*, Springer, pp. 24–39.

- [51] SPREITZER, R., AND PLOS, T. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *Constructive Side-Channel Analysis and Secure Design – COSADE* (2013), vol. 7864 of *LNCS*, Springer, pp. 200–214.
- [52] SPREITZER, R., AND PLOS, T. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In *Network and System Security – NSS* (2013), vol. 7873 of *LNCS*, Springer, pp. 656–662.
- [53] TAKAHASHI, J., FUKUNAGA, T., AOKI, K., AND FUJI, H. Highly Accurate Key Extraction Method for Access-Driven Cache Attacks Using Correlation Coefficient. In *Australasian Conference Information Security and Privacy – ACISP* (2013), vol. 7959 of *LNCS*, Springer, pp. 286–301.
- [54] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient Cache Attacks on AES, and Countermeasures. *Journal Cryptology* 23, 1 (2010), 37–71.
- [55] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems – CHES* (2003), vol. 2779 of *LNCS*, Springer, pp. 62–76.
- [56] VAN DE POL, J., SMART, N. P., AND YAROM, Y. Just a Little Bit More. In *Topics in Cryptology – CT-RSA* (2015), vol. 9048 of *LNCS*, Springer, pp. 3–21.
- [57] WEISS, M., HEINZ, B., AND STUMPF, F. A Cache Timing Attack on AES in Virtualization Environments. In *Financial Cryptography and Data Security – FC* (2012), vol. 7397 of *LNCS*, Springer, pp. 314–328.
- [58] WEISS, M., WEGGENMANN, B., AUGUST, M., AND SIGL, G. On Cache Timing Attacks Considering Multi-core Aspects in Virtualized Embedded Systems. In *Trusted Systems – INTRUST* (2014), vol. 9473 of *LNCS*, Springer, pp. 151–167.
- [59] XU, Z., BAI, K., AND ZHU, S. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-board Motion Sensors. In *Security and Privacy in Wireless and Mobile Networks – WISEC* (2012), ACM, pp. 113–124.
- [60] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014), USENIX Association, pp. 719–732.
- [61] ZHANG, K., AND WANG, X. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium* (2009), USENIX Association, pp. 17–32.

DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks

Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz *and Stefan Mangard*
Graz University of Technology, Austria

Abstract

In cloud computing environments, multiple tenants are often co-located on the same multi-processor system. Thus, preventing information leakage between tenants is crucial. While the hypervisor enforces software isolation, shared hardware, such as the CPU cache or memory bus, can leak sensitive information. For security reasons, shared memory between tenants is typically disabled. Furthermore, tenants often do not share a physical CPU. In this setting, cache attacks do not work and only a slow cross-CPU covert channel over the memory bus is known. In contrast, we demonstrate a high-speed covert channel as well as the first side-channel attack working across processors and without any shared memory. To build these attacks, we use the undocumented DRAM address mappings.

We present two methods to reverse engineer the mapping of memory addresses to DRAM channels, ranks, and banks. One uses physical probing of the memory bus, the other runs entirely in software and is fully automated. Using this mapping, we introduce DRAMA attacks, a novel class of attacks that exploit the DRAM row buffer that is shared, even in multi-processor systems. Thus, our attacks work in the most restrictive environments. First, we build a covert channel with a capacity of up to 2 Mbps, which is three to four orders of magnitude faster than memory-bus-based channels. Second, we build a side-channel template attack that can automatically locate and monitor memory accesses. Third, we show how using the DRAM mappings improves existing attacks and in particular enables practical Rowhammer attacks on DDR4.

1 Introduction

Due to the popularity of cloud services, multiple tenants sharing the same physical server through different virtual machines (VMs) is now a common situation. In

such settings, a major requirement is that no sensitive information is leaked between tenants, therefore proper isolation mechanisms are crucial to the security of these environments. While software isolation is enforced by hypervisors, shared hardware presents risks of information leakage between tenants. Previous research shows that microarchitectural attacks can leak secret information of victim processes, e.g., by clever analysis of data-dependent timing differences. Such side-channel measurements allow the extraction of secret information like cryptographic keys or enable communication over isolation boundaries via covert channels.

Cloud providers can deploy different hardware configurations, however multi-processor systems are becoming ubiquitous due to their numerous advantages. They offer high peak performance for parallelized tasks while enabling sharing of other hardware resources such as the DRAM. They also simplify load balancing while still keeping the area and cost footprint low. Additionally, cloud providers now commonly disable memory deduplication between VMs for security reasons.

To attack such configurations, successful and practical attacks must comply with the following requirements:

1. *Work across processors:* As these configurations are now ubiquitous, an attack that does not work across processors is severely limited and can be trivially mitigated by exclusively assigning processors to tenants or via the scheduler.
2. *Work without any shared memory:* With memory deduplication disabled, shared memory is not available between VMs. All attacks that require shared memory are thus completely mitigated in cross-VM settings with such configurations.

In the last years, the most prominent and well-studied example of shared-hardware exploits is cache attacks. They use the processor-integrated cache and were shown to be effective in a multitude of settings, such as cross-VM key-recovery attacks [9, 12, 20, 30], including attacks across cores [5, 14, 16, 28]. However, due to the

cache being local to the processor, these attacks do not work across processors and thus violate requirement 1. Note that in a recent concurrent work, Irazoqui et al. [11] presented a cross-CPU cache attack which exploits cache coherency mechanisms in multi-processor systems. However, their approach requires shared memory and thus violates requirement 2. The whole class of cache attacks is therefore not applicable in multi-processor systems without any shared memory.

Other attacks leverage the main memory that is a shared resource even in multi-processor systems. Xiao et al. [26] presented a covert channel that exploits memory deduplication. This covert channel has a low capacity and requires the availability of shared memory, thus violating requirement 2. Wu et al. [25] presented a covert channel exploiting the locking mechanism of the memory bus. While this attack works across processors, the capacity of the covert channel is orders of magnitude lower than that of current cache covert channels.

Therefore, only a low capacity covert channel and no side-channel have been showed with the two aforementioned requirements so far. In contrast, we demonstrate two attacks that do not use shared memory and work across processors: a high-speed covert channel as well as the first side-channel attack.

Contributions. Our attacks require knowledge of the undocumented mapping of memory addresses to DRAM channels, ranks, and banks. We therefore present two methods to reverse engineer this mapping. The first method retrieves the correct addressing functions by performing physical probing of the memory bus. The second method is entirely software-based, fully automatic, and relies only on timing differences.¹ Thus, it can be executed remotely and enables finding DRAM address mappings even in VMs in the cloud. We reverse engineered the addressing functions on a variety of processors and memory configurations. Besides consumer-grade PCs, we also analyzed a dual-CPU server system – similar to those found in cloud setups – and multiple recent smartphones.

Using this reverse-engineered mapping, we present *DRAM* attacks, a novel class of attacks that exploit the *DRAM Addressing*. In particular, they leverage DRAM row buffers that are a shared component in multi-processor systems. Our attacks require that at least one memory module is shared between the attacker and the victim, which is the case even in the most restrictive settings. In these settings, attacker and victim cannot access the same memory cells, *i.e.*, we do not circumvent system-level memory isolation. We do not make any assumptions on the cache, nor on the location of executing

¹The source code of this reverse-engineering tool and exemplary DRAMA attacks can be found at <https://github.com/IAIK/drama>.

cores, nor on the availability of shared memory such as cross-VM memory deduplication.

First, we build a covert channel that achieves transmission rates of up to 2 Mbps, which is three to four orders of magnitude faster than previously presented memory-bus based channels. Second, we build a side channel that allows to automatically locate and monitor memory accesses, *e.g.*, user input or server requests, by performing template attacks. Third, we show how the reverse-engineered mapping can be used to improve existing attacks. Existing Flush+Reload cache attacks use an incorrect cache-miss threshold, introducing noise and reducing the spatial accuracy. Knowledge of the DRAM address mapping also enables practical Rowhammer attacks on DDR4.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background information on side channels on shared hardware, on DRAM, and on the Rowhammer attack. In Section 3, we provide definitions that we use throughout the paper. In Section 4, we describe our two approaches to reverse engineer the DRAM addressing and we provide the reverse-engineered functions. In Section 5, we build a high-speed cross-CPU DRAMA covert channel. In Section 6, we build a highly accurate cross-CPU DRAMA side channel attack. In Section 7, we show how the knowledge of the DRAM addressing improves cache attacks like Flush+Reload and we show how it makes Rowhammer attacks practical on DDR4 and more efficient on DDR3. We discuss countermeasures against our attack in Section 8. We conclude in Section 9.

2 Background and related work

In this section, we discuss existing covert and side channels and give an introduction to DRAM. Furthermore, we briefly explain the Rowhammer bug and its implications.

2.1 Hardware covert and side channels

Attacks exploiting hardware sharing can be grouped into two categories. In side-channel attacks, an attacker spies on a victim and extracts sensitive information such as cryptographic keys. In covert channels however, sender and receiver are actively cooperating to exchange information in a setting where they are not allowed to, *e.g.*, across isolation boundaries.

Cache attacks. Covert and side channels using the CPU cache exploit the fact that cache hits are faster than cache misses. The methods Prime+Probe [14, 16, 19] and Flush+Reload [2, 12, 28] have been presented to either build covert or side channels. These two methods work at a different granularity: Prime+Probe can spy on cache

sets, while Flush+Reload has the finer granularity of a cache line but requires shared memory, such as shared libraries or memory deduplication.

Attacks targeting the last-level cache are cross-core, but require the sender and receiver to run on the same physical CPU. Gruss et al. [5] implemented cross-core covert channels using Prime+Probe and Flush+Reload as well as a new one, Flush+Flush, with the same protocol to normalize the results. The covert channel using Prime+Probe achieves 536 Kbps, Flush+Reload 2.3 Mbps, and Flush+Flush 3.8 Mbps. The most recent cache attack by Irazoqui et al. [11] exploits cache coherency mechanisms and work across processors. It however requires shared memory.

An undocumented function maps physical addresses to the slices of the last-level cache. However, this function has been reverse engineered in previous work [9, 15, 29], enhancing existing attacks and enabling attacks in new environments.

Memory and memory bus. Xiao et al. [26] presented a covert channel that exploits memory deduplication. In order to save memory, the hypervisor searches for identical pages in physical memory and merges them across VMs to a single read-only physical page. Writing to this page triggers a copy-on-write page fault, incurring a significantly higher latency than a regular write access. The authors built a covert channel that achieves up to 90 bps, and 40 bps on a system under memory pressure. Wu et al. [25] proposed a bus-contention-based covert channel, that uses atomic memory operations locking the memory bus. This covert channel achieves a raw bandwidth of 38 Kbps between two VMs, with an effective capacity of 747 bps with error correction.

2.2 DRAM organization

Modern DRAM is organized in a hierarchy of channels, DIMMs, ranks, and banks. A system can have one or more *channels*, which are physical links between the DRAM modules and the memory controller. Channels are independent and can be accessed in parallel. This allows distribution of the memory traffic, increasing the bandwidth, and reducing the latency in many cases. Multiple *Dual Inline Memory Modules (DIMMs)*, which are the physical memory modules attached to the mainboard, can be connected to each channel. A DIMM typically has one or two *ranks*, which often correspond to the front and back of the physical module. Each rank is composed of *banks*, typically 8 on DDR3 DRAM and 16 on DDR4 DRAM. In the case of DDR4, banks are additionally grouped into *bank groups*, e.g., 4 bank groups with 4 banks each. Banks finally contain the actual memory arrays which are organized in *rows* (typically 2^{14} to 2^{17}) and *columns* (often 2^{10}). On PCs, the DRAM word size

and bus width is 64 bits, resulting in a typical row size of 8 KB. As channel, rank and bank form a hierarchy, two addresses can only be physically adjacent in the DRAM chip if they are in the same channel, DIMM, rank and bank. In this case we just use the term same bank.

The memory controller, which is integrated into modern processors, translates physical addresses to channels, DIMMs, ranks, and banks. AMD publicly documents the addressing function used by its products (see, e.g., [1, p. 345]), however to the best of our knowledge Intel does not. The mapping for one Intel Sandy Bridge machine in one memory configuration has been reverse engineered by Seaborn [23]. However, Intel has changed the mapping used in its more recent microarchitectures. Also, the mapping necessarily differs when using other memory configurations, e.g., a different number of DIMMs.

The row buffer. Apart from the memory array, each bank also features a row buffer between the DRAM cells and the memory bus. From a high-level perspective, it behaves like a directly-mapped cache and stores an entire DRAM row. Requests to addresses in the currently active row are served directly from this buffer. If a different row needs to be accessed, then the currently active row is first closed (with a pre-charge command) and then the new row is fetched (with a row-activate command). We call such an event a row conflict. Naturally, such a conflict leads to significantly higher access times compared to requests to the active row. This timing difference will later serve as the basis for our attacks and for the software-based reverse-engineering method. Note that after each refresh operation, a bank is already in the pre-charged state. In this case, no row is currently activated.

Independently of our work, Hassan et al. [7] also proposed algorithms to reverse engineer DRAM functions based on timing differences. However, their approach requires customized hardware performance-monitoring units. Thus, they tested their approach only in a simulated environment and not on real systems. Concurrently to our work, Xiao et al. [27] proposed a method to reverse engineer DRAM functions based on the timing differences caused by row conflicts. Although their method is similar to ours, their focus is different, as they used the functions to then perform Rowhammer attacks across VMs.

DRAM organization for multi-CPU systems. In modern multi-CPU server systems, each CPU features a dedicated memory controller and attached memory. The DRAM is still organized in one single address space and is accessible by all processors. Requests for memory attached to other CPUs are sent over the CPU interconnect, e.g., Intel’s QuickPath Interconnect (QPI). This memory design is called Non-Uniform Memory Access (NUMA), as the access time depends on the memory location.

On our dual Haswell-EP setup, the organization of this single address space can be configured for the expected workload. In *interleaved mode*, the memory is split into small slices which are spliced together in an alternating fashion. In *non-interleaved* mode, each CPUs memory is kept in one contiguous physical-address block. For instance, the lower half of the address space is mapped to the first CPUs memory, whereas the upper half is mapped to the second CPUs memory.

2.3 The Rowhammer bug

The increasing DRAM density has led to physically smaller cells, which can thus store smaller charges. As a result, the cells have a lower noise margin and the level of parasitic electrical interaction is potentially higher, resulting in the so-called Rowhammer bug [8, 13, 18].

This bug results in corruption of data, not in rows that are directly accessed, but rather in adjacent ones. When performing random memory accesses, the probability for such faults is virtually zero. However, it rises drastically when performing accesses in a certain pattern. Namely, flips can be caused by frequent activation (*hammering*) of adjacent rows. As data needs to be served from DRAM and not the cache, an attack needs to either flush data from the cache using the `clflush` instruction in native environments [13], or using cache eviction in other more restrictive environments, e.g., JavaScript [4].

Seaborn [22] implemented two attacks that exploit the Rowhammer bug, showing the severity of faulting single bits for security. The first exploit is a kernel privilege escalation on a Linux system, caused by a bit flip in a page table entry. The second one is an escape of Native Client sandbox caused by a bit flip in an instruction sequence for indirect jumps.

3 Definitions

In this section we provide definitions for the terms *row hit* and *row conflict*. These definitions provide the basis for our reverse engineering as well as the covert and side channel attacks.

Every physical memory location maps to one out of many rows in one out of several banks in the DRAM. Considering a single access to a row i in a bank there are two major possible cases:

1. The row i is already opened in the row buffer. We call this case a *row hit*.
2. A different row $j \neq i$ in the same bank is opened. We call this case a *row conflict*.

Considering frequent alternating accesses to two (or more) addresses we distinguish three cases:

1. The addresses map to different banks. In this case the accesses are independent and whether the ad-

dresses have the same row indices has no influence on the timing. Row hits are likely to occur for the accesses, *i.e.*, access times are low.

2. The addresses map to the same row i in the same bank. The probability that the row stays open in between accesses is high, *i.e.*, access times are low.
3. The addresses map to the different rows $i \neq j$ in the same bank. Each access to an address in row i will close row j and vice versa. Thus, row conflicts occur for the accesses, *i.e.*, access times are high.

To measure the timing differences of row hits and row conflicts, data has to be flushed from the cache. Figure 1 shows a comparison of standard histograms of access times for cache hits and cache misses. Cache misses are further divided into row hits and row conflicts. For this purpose an unrelated address in the same row was accessed to cause a row hit and an unrelated address in the same bank but in a different row was accessed to cause a row conflict. We see that from 180 to 216 cycles row hits occur, but no row conflicts (cf. highlighted area in Figure 1). In the remainder, we build different attacks that are based on this timing difference between row hits and row conflicts.

4 Reverse engineering DRAM addressing

In this section, we present our reverse engineering of the DRAM address mapping. We discuss two approaches, the first one is based on physical probing, whereas the second one is entirely software-based and fully automated. Finally, we present the outcome of our analysis, *i.e.*, the reverse-engineered mapping functions. In the remainder of this paper, we denote with a a physical memory address. a_i denotes the i -th bit of an address.

4.1 Linearity of functions

The DRAM addressing functions are reverse engineered in two phases. First, a measuring phase and second, a subsequent solving phase. Our solving approaches require that the addressing functions are linear, *i.e.*, they are XORs of physical-address bits.

In fact, Intel used such functions in earlier microarchitectures. For instance, Seaborn [23] reports that on his Sandy Bridge setup the bank address is computed by XORing the bits $a_{14..16}$ with the lower bits of the row number ($a_{18..20}$) (cf. Figure 4a). This is done in order to minimize the number of row conflicts during runtime. Intel also uses linear functions for CPU-cache addressing. Maurice et al. [15] showed that the *complex addressing* function, which is used to select cache slices, is an XOR of many physical-address bits.

As it turns out, linearity holds on all our tested configurations. However, there are setups in which it might be

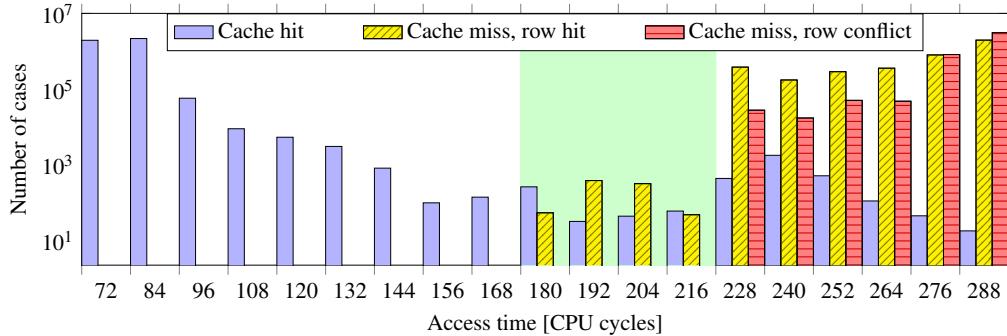


Figure 1: Histogram for cache hits and cache misses divided into row hits and row conflicts on the Ivy Bridge i5 test system. Measurements were performed after a short idle period to simulate non-overlapping accesses by victim and spy. From 180 to 216 cycles row hits occur, but no row conflicts.

violated, such as triple-channel configurations. We did not test such systems and leave a reverse engineering to future work.

4.2 Reverse engineering using physical probing

Our first approach to reverse engineer the DRAM mapping is to physically probe the memory bus and to directly read the control signals. As shown in Figure 2, we use a standard passive probe to establish contact with the pin at the DIMM slot. We then repeatedly accessed a selected physical address² and used a high-bandwidth oscilloscope to measure the voltage and subsequently deduce the logic value of the contacted pin. Note that due to the repeated access to a single address, neither a timely location of specific memory requests nor distinguishing accesses to the chosen address from other random ones is required.

We repeated this experiment for many selected addresses and for all pins of interest, namely the bank-address bits (BA0, BA1, BA2 for DDR3 and BG0, BG1, BA0, BA1 for DDR4) for one DIMM and the chip select CS for half the DIMMs.

For the solving phase we use the following approach. Starting from the top-layer (channel or CPU addressing) and drilling down, for each DRAM addressing function we create an over-defined system of linear equations in the physical address bits. The left-hand-side of this system is made up of the relevant tested physical addresses. For instance, for determining the bank functions we only use addresses that map to the contacted DIMMs channel. The right-hand-side of the system of equations are the previously measured logic values for the respective

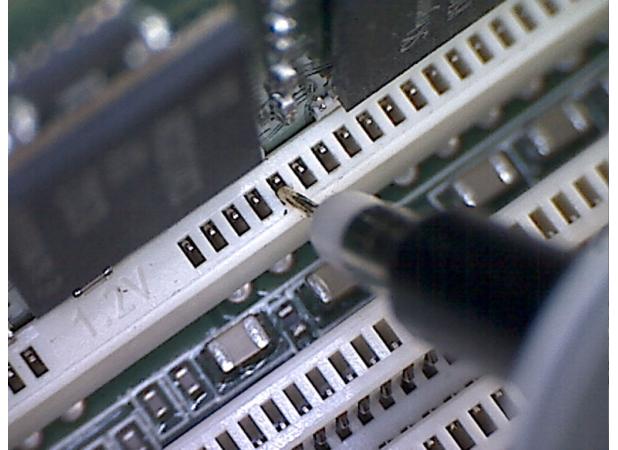


Figure 2: Physical probing of the DIMM slot.

address and the searched-for function. The logic values for CPU and channel addressing are computed by simply ORing all respective values for the chip-select pins. We then solve this system using linear algebra. The solution is the corresponding DRAM addressing function.

Obviously, this reverse-engineering approach has some drawbacks. First, expensive measurement equipment is needed. Second, it requires physical access to the internals of the tested machine. However, it has the big advantage that the address mapping can be reconstructed for each control signal individually and exactly. Thus, we can determine the exact individual functions for the bus pins. Furthermore, every platform only needs to be measured only once in order to learn the addressing functions. Thus, an attacker does not need physical access to the concrete attacked system if the measurements are performed on a similar machine.

²Resolving virtual to physical addresses requires root privileges in Linux. Given that we need physical access to the internals of the system, this is a very mild prerequisite.

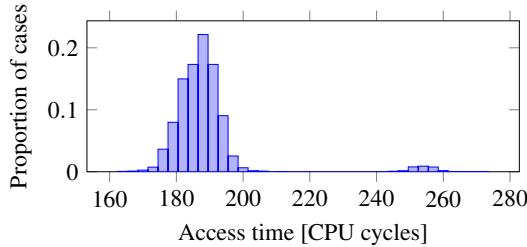


Figure 3: Histogram of average memory access times for random address pairs on our Haswell test system. A clear gap separates the majority of address pairs causing no row conflict (lower access times), because they map to different banks, from the few address pairs causing a row conflict (higher access times), because they map to different rows in the same bank.

4.3 Fully automated reverse engineering

For our second approach to reverse engineer the DRAM mapping we exploit the fact that row conflicts lead to higher memory access times. We use the resulting timing differences to find sets of addresses that map to the same bank but to a different row. Subsequently, we determine the addressing functions based on these sets. The entire process is fully automated and runs in unprivileged and possibly restricted environments.

Timing analysis. In the first step, we aim to find same-bank addresses in a large array mapped into the attackers' address space. For this purpose, we perform repeated alternating access to two addresses and measure the average access time. We use `clflush` to ensure that each access is served from DRAM and not from the CPU cache. As shown in Figure 3, for some address pairs the access time is significantly higher than for most others. These pairs belong to the same bank but to different rows. The alternating access causes frequent row conflicts and consequently the high latency.

The tested pairs are drawn from an address pool, which is built by selecting random addresses from a large array. A small subset of addresses in this pool is tested against all others in the pool. The addresses are subsequently grouped into sets having the same channel, DIMM, rank, and bank. We try to identify as many such sets as possible in order to reconstruct the addressing functions.

Function reconstruction. In the second phase, we use the identified address sets to reconstruct the addressing functions. This reconstruction requires (at least partial) resolution of the tested virtual addresses to physical ones. Similar as later in Section 5.1, one can use either the availability of 2 MB pages, 1 GB pages, or privileged information such as the virtual-to-physical address transla-

tion that can be obtained through `/proc/pid/pagemap` in Linux systems.

In the case of 2 MB pages we can recover all partial functions up to bit a_{20} , as the lowest 21 bit of virtual and physical address are identical. On many systems the DRAM addressing functions do not use bits above a_{20} or only few of them, providing sufficient information to mount covert and side-channel attacks later on. In the case of 1 GB pages we can recover all partial functions up to bit a_{30} . This is sufficient to recover the full DRAM addressing functions on all our test systems. If we have full access to physical address information we will still ignore bits a_{30} and upwards. These bits are typically only used for DRAM row addressing and they are very unlikely to play any role in bank addressing. Additionally, we ignore bits $(a_0..a_5)$ as they are used for addressing within a cache line.

The search space is then small enough to perform a brute-force search of linear functions within seconds. For this, we generate all linear functions that use exactly n bits as coefficients and then apply them to all addresses in one randomly selected set. We start with $n = 1$ and increment n subsequently to find all functions. Only if the function has the same result for all addresses in a set, we test this potential function on all other sets. However, in this case we only pick one address per set and test whether the function is constant over all sets. If so, the function is discarded. We obtain a list of possible addressing functions that also contains linear combinations of the actual DRAM addressing functions. We prioritize functions with a lower number of coefficients, *i.e.*, we remove higher-order functions which are linear combinations of lower-order ones. Depending on the random address selection, we now have a complete set of correct addressing functions. We verify the correctness either by comparing it to the results from the physical probing, or by performing a software-based test, *i.e.*, verifying the timing differences on a larger set of addresses, or verifying that usage of the addressing functions in Rowhammer tests increases the number of bit flips per second by a factor that is the number of sets we found.

Compared to the probing approach, this purely software-based method has significant advantages. It does not require any additional measurement equipment and can be executed on a remote system. We can identify the functions even from within VMs or sandboxed processes if 2 MB or 1 GB pages are available. Furthermore, even with only 4 KB pages we can group addresses into sets that can be directly used for covert or side channel attacks. This software-based approach also allows reverse engineering in settings where probing is not easily possible anymore, such as on mobile devices with hard-wired ball-grid packages. Thus, it allowed us to reverse engineer the mapping on current ARM processors.

Table 1: Experimental setups.

CPU / SoC	Microarch.	Mem.
i5-2540M	Sandy Bridge	DDR3
i5-3230M	Ivy Bridge	DDR3
i7-3630QM	Ivy Bridge	DDR3
i7-4790	Haswell	DDR3
i7-6700K	Skylake	DDR4
2x Xeon E5-2630 v3	Haswell-EP	DDR4
Qualcomm Snapdragon S4 Pro	ARMv7	LPDDR2
Samsung Exynos 5 Dual	ARMv7	LDDR3
Qualcomm Snapdragon 800	ARMv7	LPDDR3
Qualcomm Snapdragon 820	ARMv8-A	LPDDR3
Samsung Exynos 7420	ARMv8-A	LPDDR4

One downside of the software-based approach is that it cannot recover the exact labels (BG0, BA0, ...) of the functions. Thus, we can only guess whether the reconstructed function computes a bank address bit, rank bit, or channel bit. Note that assigning the correct labels to functions is not required for any of our attacks.

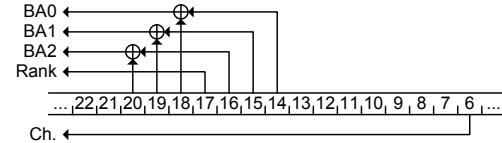
4.4 Results

We now present the reverse-engineered mappings for all our experimental setups. We analyzed a variety of systems (Table 1), including a dual-CPU Xeon system, that can often be found in cloud systems, and multiple current smartphones. Where possible, we used both presented reverse-engineering methods and cross-validated the results.

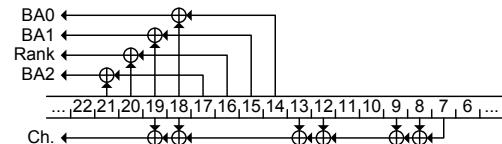
We found that the basic scheme is always as follows. On PCs, the memory bus is 64 bits wide, yet the smallest addressable unit is a byte. Thus, the three lower bits ($a_0..a_2$) of the physical address are used as byte index into a 64-bit (8-byte) memory word and they are never transmitted on the memory bus. Then, the next bits are used for column selection. One bit in between is used for channel addressing. The following bits are responsible for bank, rank, and DIMM addressing. The remaining upper bits are used for row selection.

The detailed mapping, however, differs for each setup. To give a quick overview of the main differences, we show the mapping of one selected memory configuration for multiple Intel microarchitectures and ARM-based SoCs in Figure 4. Here we chose a configuration with two equally sized DIMMs in dual-channel configuration, as it is found in many off-the-shelf consumer PCs. All our setups use dual-rank DIMMs and use 10 bits for column addressing. Figure 4a shows the mapping on the Sandy Bridge platform, as reported by Seaborn [23]. Here, only a_6 is used to select the memory channel, a_{17} is used for rank selection. The bank-address bits are com-

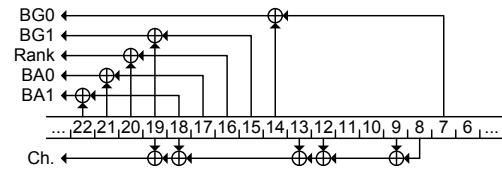
puted by XORing bits $a_{14..a_{16}}$ with the lower bits of the row index ($a_{18..a_{20}}$).



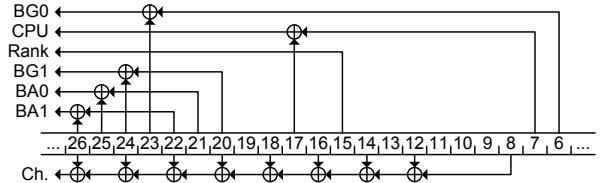
(a) Sandy Bridge – DDR3 [23].



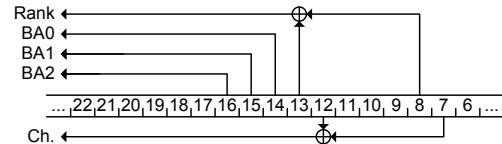
(b) Ivy Bridge / Haswell – DDR3.



(c) Skylake – DDR4.



(d) Dual Haswell-EP (Interleaved Mode) – DDR4.



(e) Samsung Exynos 7420 – LPDDR4.

Figure 4: Reverse engineered dual channel mapping (1 DIMM per channel) for different architectures.

The channel selection function changed with later microarchitectures, such as Ivy Bridge and Haswell. As shown in Figure 4b, the channel-selection bit is now computed by XORing seven bits of the physical address. Further analysis showed that bit a_7 is used exclusively, i.e., it is not used as part of the row- or column address.

Additionally, rank selection is now similar to bank addressing and also uses XORs.

Our Skylake test system uses DDR4 instead of DDR3. Due to DDR4's introduction of bank grouping and the doubling of the available banks (now 16), the addressing function necessarily changed again. As shown in Figure 4c, a_7 is not used for channel selection anymore, but for bank addressing instead.

Figure 4d depicts the memory mapping of a dual-CPU Haswell-EP system equipped with DDR4 memory. It uses 2 modules in dual-channel configuration *per CPU* (4 DIMMs in total). In interleaved mode (cf. Section 2.2), the chosen CPU is determined as $a_7 \oplus a_{17}$. Apart from the different channel function, there is also a difference in the bank addressing, *i.e.*, bank addressing bits are shifted. The range of bits used for row indexing is now split into address bits ($a_{17..a_{19}}$) and a_{23} upwards.

The mapping used on one of our mobile platforms, a Samsung Galaxy S6 with an Exynos 7420 ARMv8-A SoC and LPDDR4 memory, is much simpler (cf. Figure 4e). Here physical address bits are mapped directly to bank address bits. Rank and channel are computed with XORs of only two bits each. The bus width of LPDDR4 is 32 bits, so only the two lowest bits are used for byte indexing in a memory word.

Table 2 shows a comprehensive overview of all platforms and memory configurations we analyzed. As all found functions are linear, we simply list the index of the physical address bits that are XORed together. With the example of the Haswell microarchitecture, one can clearly see that the indices are shifted to accommodate for the different memory setups. For instance, in single-channel configurations a_7 is used for column instead of channel selection, which is why bank addressing starts with a_{13} instead of a_{14} .

5 A high-speed cross-CPU covert channel

In this section, we present a first DRAMA attack, namely a high-speed cross-CPU covert channel that does not require shared memory. Our channel exploits the row buffer, which behaves like a directly-mapped cache. Unlike cache attacks, the only prerequisite is that two communicating processes have access to the same memory module.

5.1 Basic concept

Our covert channel exploits timing differences caused by row conflicts. Sender and receiver occupy different rows in the same bank as illustrated in Figure 5. The receiver process continuously accesses a chosen physical address in the DRAM and measures the average access time over a few accesses. If the sender process now continuously

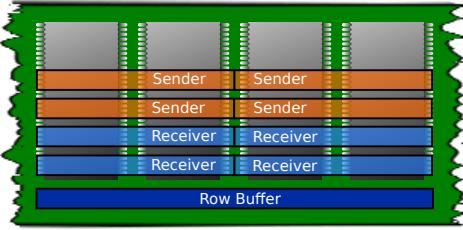
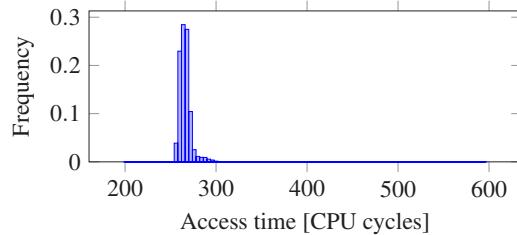
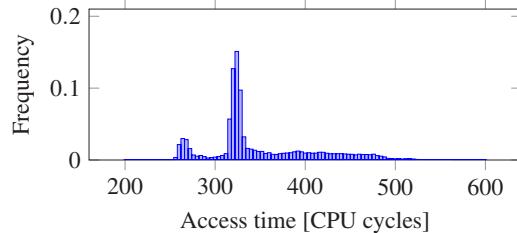


Figure 5: The sender occupies rows in a bank to trigger row conflicts. The receiver occupies rows in the same bank to observe these row conflicts.



(a) Sender inactive on bank: sending a 0.



(b) Sender active on bank: sending a 1.

Figure 6: Timing differences between active and non-active sender (on one bank), measured on the Haswell i7 test system.

accesses a different address in the same bank but in a different row, a row conflict occurs. This leads to higher average access times in the receiver process. Bits can be transmitted by switching the activity of the sender process in the targeted bank on and off. This timing difference is illustrated in Figure 6, an exemplary transmission is shown in Figure 7. The receiver process distinguishes the two values based on the mean access time. We assign a logic value of 0 to low access times (the sender is inactive) and a value of 1 to high access times (the sender is active).

Each (CPU, channel, DIMM, rank, bank) tuple can be used as a separate transmission channel. However, a high number of parallel channels leads to increased noise. Also, there is a strict limit on the usable bank par-

Table 2: Reverse engineered DRAM mapping on all platforms and configurations we analyzed via physical probing or via software analysis. These tables list the bits of the physical address that are XORed. For instance, for the entry (13, 17) we have $a_{13} \oplus a_{17}$.

(a) DDR3									
CPU	Ch.	DIMM/Ch.	BA0	BA1	BA2	Rank	DIMM	Channel	
Sandy Bridge	1	1	13, 17	14, 18	15, 19	16	-	-	
Sandy Bridge [23]	2	1	14, 18	15, 19	16, 20	17	-	6	
	1	1	13, 17	14, 18	16, 20	15, 19	-	-	
Ivy Bridge/Haswell	1	2	13, 18	14, 19	17, 21	16, 20	15	-	
	2	1	14, 18	15, 19	17, 21	16, 20	-	7, 8, 9, 12, 13, 18, 19	
	2	2	14, 19	15, 20	18, 22	17, 21	16	7, 8, 9, 12, 13, 18, 19	

(b) DDR4									
CPU	Ch.	DIMM/Ch.	BG0	BG1	BA0	BA1	Rank	CPU	Channel
Skylake [†]	2	1	7, 14	15, 19	17, 21	18, 22	16, 20	-	8, 9, 12, 13, 18, 19
2x Haswell-EP	1	1	6, 22	19, 23	20, 24	21, 25	14	7, 17	-
(interleaved)	2	1	6, 23	20, 24	21, 25	22, 26	15	7, 17	8, 12, 14, 16, 18, 20, 22, 24, 26
2x Haswell-EP	1	1	6, 21	18, 22	19, 23	20, 24	13	-	-
(non-interleaved)	2	1	6, 22	19, 23,	20, 24	21, 25	14	-	7, 12, 14, 16, 18, 20, 22, 24, 26

(c) LPDDR2,3,4						
CPU	Ch.	BA0	BA1	BA2	Rank	Channel
Qualcomm Snapdragon S4 Pro [†]	1	13	14	15	10	-
Samsung Exynos 5 Dual [†]	1	13	14	15	7	-
Qualcomm Snapdragon 800/820 [†]	1	13	14	15	10	-
Samsung Exynos 7420 [†]	2	14	15	16	8, 13	7, 12

[†] Software analysis only. Labeling of functions is based on results of other platforms.

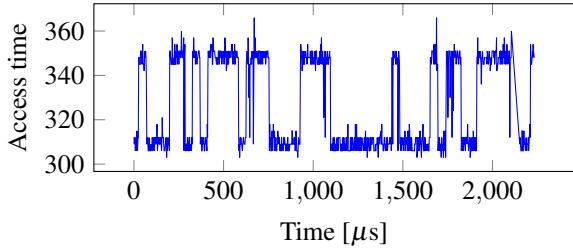


Figure 7: Covert channel transmission on one bank, cross-CPU and cross-VM on a Haswell-EP server. The time frame for one bit is 50 μ s.

allelism. Thus, optimal performance is achieved when using only a subset of available tuples. Transmission channels are unidirectional, but the direction can be chosen for each one independently. Thus, two-way communication is possible.

To evaluate the performance of this new covert channel, we created a proof-of-concept implementation. We restrict ourselves to unidirectional communication, *i.e.*, there is one dedicated sender and one dedicated receiver.

The memory access time is measured using `rdtsc`. The memory accesses are performed using volatile pointers. In order to cause a DRAM access for each request, data has to be flushed from the cache using `clflush`.

Determining channel, rank, and bank address. In an agreement phase, all parties need to agree on the set of (channel, DIMM, rank, bank) tuples that are used for communication. This set needs to be chosen only once, all subsequent communication can use the same set. Next, both sender and receiver need to find at least one address in their respective address space for each tuple. Note that some operating systems allow unprivileged resolution of virtual to physical addresses. In this case, finding correct addresses is trivial.

However, on Linux, which we used on our testing setup, unprivileged address resolution is not possible. Thus, we use the following approach. As observed in previous work [3, 4], system libraries and the operating system assign 2 MB pages for arrays which are significantly larger than 2 MB. On these pages, the 21 lowest bits of the virtual address and the physical address are

identical. Depending on the hardware setup, these bits can already be sufficient to fully determine bank, rank and channel address. For this purpose, both processes request a large array. The start of this array is not necessarily aligned with a 2 MB border. Memory before such a border is allocated using 4 KB pages. We skip to the next 2 MB page border by choosing the next virtual address having the 21 lowest bits set to zero.

On systems that also use higher bits, an attacker can use the following approach, which we explain on the example of the mapping shown in Figure 4b. There an attacker cannot determine the BA2 bit by just using 2 MB pages. Thus, the receiving process selects addresses with chosen BA0, BA1, rank, and channel, but unknown BA2 bit. The sender now accesses addresses for both possibilities of BA2, e.g., by toggling a_{17} between consecutive reads. Thus, only each second access in the sending process targets the correct bank. Yet, due to bank parallelism this does not cause a notable performance decrease. Note however that this approach might not work if the number of unknown bank-address bits is too high.

In a virtualized environment, even a privileged attacker is able to retrieve only the *guest physical address*, which is further translated into the real physical address by the memory management unit. However, if the host system uses 1 GB pages for the second-level address translation (to improve efficiency), then the lowest 30 bits of the guest physical address are identical to the real physical address. Knowledge of these bits is sufficient on all systems we analyzed to use the full DRAM addressing functions.

Finally, the covert channel could also be built without actually reconstructing the DRAM addressing functions. Instead of determining the exact bank address, it can rely solely on the same-bank sets retrieved in Section 4.3. In an initialization phase, both sender and receiver perform the timing analysis and use it to build sets of same-bank addresses. Subsequently, the communicating parties need to synchronize their sets, *i.e.*, they need to agree on which of them is used for transmission. This is done by sending predefined patterns over the channel. After that, the channel is ready for transmission. Thus, it can be established without having any information on the mapping function nor on the physical addresses.

Synchronization. In our proof-of-concept implementation, one set of bits (a data block) is transmitted for a fixed time span which is agreed upon before starting communication. Decreasing this period increases the raw bitrate, but it also increases the error rate, as shown in Figure 8.

For synchronizing the start of these blocks we employ two different mechanisms. If sender and receiver run natively, we use the wall clock as means of synchronization. Here blocks start at fixed points in time. If,

however, sender and receiver run in two different VMs, then a common (or perfectly synchronized) wall clock is typically not available. In this case, the sender uses one of the transmission channels to transmit a clock signal which toggles at the beginning of each block. The receiver then recovers this clock and can thus synchronize with the sender.

We employ multiple threads for both the sender and receiver processes to achieve optimal usage of the memory bus. Thus, memory accesses are performed in parallel, increasing the performance of the covert channel.

5.2 Evaluation

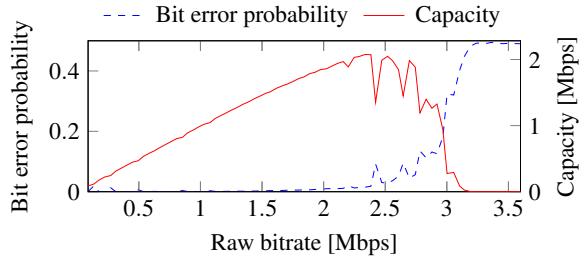
We evaluated the performance of our covert-channel implementation on two systems. First, we performed tests on a standard desktop PC featuring an Intel i7-4790 CPU with Haswell microarchitecture. It was equipped with 2 Kingston DDR3 KVR16N11/8 dual-rank 8 GB DIMMs in dual-channel configuration. The system was mostly idle during the tests, *i.e.*, there were no other tasks causing significant load on the system. The DRAM clock was set to its default of 800 MHz (DDR3-1600).

Furthermore, we also tested the capability of cross-CPU transmission on a server system. Our setup has two Intel Xeon E5-2630 v3 (Haswell-EP microarchitecture). It was equipped with a total of 4 Samsung M393A2G40DB0-CPB DDR4 registered ECC DIMMs. Each CPU was connected to two DIMMs in dual-channel configuration and NUMA was set to interleaved mode. The DRAM frequency was set to its maximum supported value (DDR4-1866).

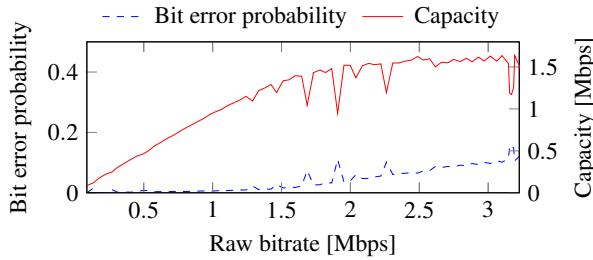
For both systems, we evaluated the performance in both a native scenario, *i.e.*, both processes run natively, and in a cross-VM scenario. We transmit 8 bits per block (use 8 (CPU, channel, DIMM, rank, bank) tuples) in the covert channel and run 2 threads in both the sender and the receiver process. Every thread is scheduled to run on different CPU cores, and in the case of the Xeon system, sender and receiver run on different physical CPUs.

We tested our implementation with a large range of measurement intervals. For each one, we measure the raw channel capacity and the bit error probability. While the raw channel capacity increases proportionally to the reduction of the measurement time, the bit error rate increases significantly if the measurement time is too short. In order to find the best transmission rate, we use the channel capacity as metric. When using the binary symmetric channel model, this metric is computed by multiplying the raw bitrate with $1 - H(e)$, with e the bit error probability and $H(e) = -e \cdot \log_2(e) - (1 - e) \cdot \log_2(1 - e)$ the binary entropy function.

Figure 8 shows the error rate varying depending on the raw bitrate for the case that both sender and receiver



(a) Desktop setup (Haswell)



(b) Server setup, cross-CPU (Haswell-EP)

Figure 8: Performance of our covert channel implementation (native).

run natively. On our desktop setup (Figure 8a), the error probability stays below 1% for bitrates of up to 2 Mbps. The channel capacity reaches up to 2.1 Mbps (raw bitrate of 2.4 Mbps, error probability of 1.8%). Beyond this peak, the increasing error probability causes a decrease in the effective capacity. On our server setup (Figure 8b) the cross-CPU communication achieves 1.2 Mbps with a 1% error rate. The maximum capacity is 1.6 Mbps (raw 2.6 Mbps, 8.7% error probability).

For the cross-core cross-VM scenario, we deployed two VMs which were configured to use 1 GB pages for second-stage address translation. We reach a maximum capacity of 309 kbps (raw 411 kbps, 4.1% error probability) on our desktop system. The server setup (cross-CPU cross-VM) performs much better, we achieved a bitrate of 596 kbps with an error probability of just 0.4%.

5.3 Comparison with state of the art

We compare the bitrate of our DRAM covert channel with the normalized implementation of three cache covert channels by Gruss et al. [5]. For an error rate that is less than 1%, the covert channel using Prime+Probe obtains 536 Kbps, the one using Flush+Reload 2.3 Mbps and the one using Flush+Flush 3.8 Mbps. With a capacity of up to 2 Mbps, our covert channel is within the same order of magnitude of current cache-based channels. However, unlike Flush+Reload and Flush+Flush, it

does not require shared memory. Moreover, in contrast to our attack, these cache covert channels do not allow cross-CPU communication.

The work of Irazoqui et al. [11] focuses on cross-CPU cache-based side-channel attacks. They did not implement a covert channel, thus we cannot compare our performance with their cache attack. However, their approach also requires shared memory and thus it would not work in our attack setting.

The covert channel by Xiao et al. [26] using memory deduplication achieves up to 90 bps. However, due to security concerns, memory deduplication has been disabled in many cloud environments. The covert channel of Wu et al. [25] using the memory bus achieves 746 bps with error correction. Our covert channel is therefore three to four orders of magnitude faster than state-of-the-art memory-based covert channels.

6 A low-noise cross-CPU side channel

In this section, we present a second DRAMA attack, a highly accurate side-channel attack using DRAM addressing information. We again exploit the row buffer and its behavior similar to a directly-mapped cache. In this attack, the spy and the victim can run on separate CPUs and do not share memory, *i.e.*, no access to shared libraries and no page deduplication between VMs. We mainly consider a local attack scenario where Flush+Reload cache attacks are not applicable due to the lack of shared memory. However, our side-channel attacks can also be applied in a cloud scenario where multiple users on a server and one malicious user spies on other users through this side channel. The side channel achieves a timing accuracy that is comparable to Flush+Reload and a higher spatial accuracy than Prime+Probe. Thus, it can be used as a highly accurate alternative to Prime+Probe cache attacks in cross-core scenarios without shared memory.

6.1 Basic concept

In case of the covert channel, an active sender caused row conflicts. In the side-channel attack, we infer the activity of a victim process by detecting *row hits* and *row conflicts* following our definitions from Section 3. For the attack to succeed, spy and victim need to have access to the same row in a bank, as illustrated in Figure 9. This is possible without shared memory due to the DRAM addressing functions.

Depending on the addressing functions, a single 4 KB page can map to multiple DRAM rows. As illustrated in Figure 10, in our Haswell-EP system the contents of a page are split over 8 DRAM rows (with the same row index, but different bank address). Conversely, a

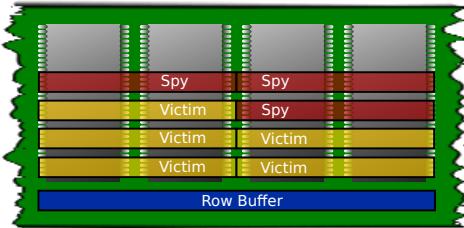


Figure 9: Victim and spy have memory allocated in the same DRAM row. By accessing this memory, the spy can determine whether the victim just accessed it.

DRAM row contains content of at least two 4 KB pages, as the typical row size is 8 KB. More specifically, in our Haswell-EP setup a single row stores content for 16 different 4 KB pages, as again shown in Figure 10. The amount of memory mapping from one page to one specific row, e.g., 512 bytes in the previous case, is the achievable spatial accuracy of our attack. If none of the DRAM addressing functions uses low address bits ($a_0 - a_{11}$), the spatial accuracy is 4 KB, which is the worst case. However, if DRAM addressing functions (channel, BG0, CPU, etc.) use low address bits, a better accuracy can be achieved, such as the 512 B for the server setup. On systems where 6 or more low address bits are used, the spatial accuracy of the attack is 64 B and thus as accurate as a Flush+Reload cache side-channel attack.

Assuming that an attacker occupies at least one other 4 KB page that maps (in part) to the same bank and row, the attacker has established a situation as illustrated in Figure 9.

To run the side-channel attack on a private memory address t in a victim process, the attacker allocates a memory address p that maps to the same bank and the same row as the target address t . As shown in Figure 10, although t and p map to the same DRAM row, they belong to different 4 KB pages (*i.e.*, no shared memory). The attacker also allocates a row conflict address \bar{p} that maps to the same bank but to a different row.

The side-channel attack then works in three steps:

1. Access the row conflict address \bar{p}
2. Wait for the victim to compute
3. Measure the access time on the targeted address p

If the measured timing is below a row-hit threshold (cf. the highlighted ‘‘row hit’’ region in Figure 1), the victim has just accessed t or another address in the target row. Thus, we can accurately determine when a specific non-shared memory location is accessed by a process running on another core or CPU. As p and \bar{p} are on separate private 4 KB pages, they will not be prefetched and we can measure row hits without any false positives. By allocat-

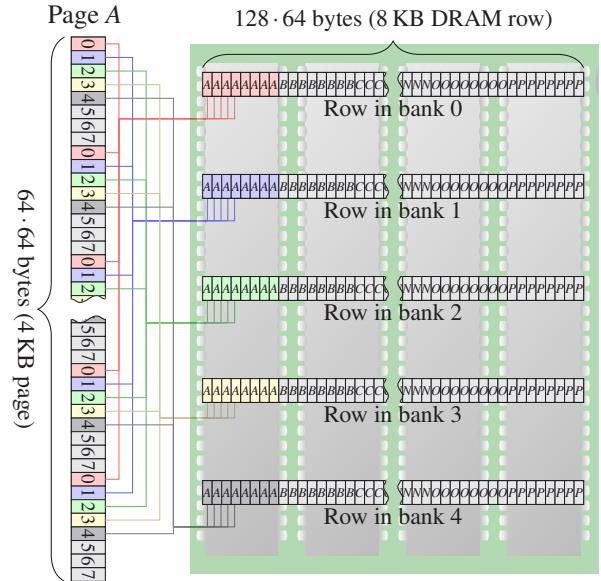


Figure 10: Mapping between a 4 KB page and an 8 KB DRAM row in the Haswell-EP setup. Banks are numbered 0 – 7, pages are numbered $A - P$. Every eighth 64-byte region of a 4 KB page maps to the same bank in DRAM. In total 8 out of 64 regions (= 512 B) map to the same bank. Thus, the memory of each row is divided among 16 different pages ($A - P$) that use memory from the same row. Occupying one of the pages $B - P$ is sufficient to spy on the eight 64-byte regions of page A in the same bank.

ing all but one of the pages that map to a row, the attacker maximizes the spatial accuracy.

Based on this attack principle, we build a fully automated template attack [6] that triggers an event in the victim process running on the other core or CPU (*e.g.*, by sending requests to a web interface or triggering user-interface events). For this attack we do not need to reconstruct the full addressing functions nor determine the exact bank address. Instead, we exploit the timing difference between row hits and row conflicts as shown in Figure 1.

To perform a DRAMA template attack, the attacker allocates a large fraction of memory, ideally in 4 KB pages. This ensures that some of the allocated pages are placed in a row together with pages used by the victim. The attacker then profiles the entire allocated memory and records the row-hit ratio for each address.

False positive detections are eliminated by running the profiling phase with different events. If an address has a high row-hit ratio for a single event, it can be used to monitor that event in the exploitation phase. After such an address has been found, all other remaining mem-

ory pages will be released and the exploitation phase is started.

6.2 Evaluation

We evaluated the performance of our side-channel attack in several tests. These tests were performed on a dual-core laptop with an Ivy Bridge Intel i5-3230M CPU with 2 Samsung DDR3-1600 dual-rank 4 GB DIMMs in dual-channel configuration.

The first test was a DRAMA template attack. The attack ran without any shared memory in an unprivileged user program. In this template attack we profiled access times on a private memory buffer while triggering keystrokes in the Firefox address bar. Figure 11 shows the template attack profile with and without keystrokes being triggered. While scanning a total of 7 GB of allocated memory, we found 1195 addresses that showed at least one row hit during the tests. 59 of these addresses had row hits independent of the event (false positives), *i.e.*, these 59 addresses cannot be used to monitor keystroke events. For the remaining 1136 addresses we only had row hits after triggering a keystroke in the Firefox address bar. Out of these addresses, 360 addresses had more than 20 row hits. Any of these 360 addresses can be used to monitor keystrokes reliably. The time to find an exploitable address varies between a few seconds and multiple minutes. Sometimes the profiling phase does not find any exploitable address, for instance if there is no memory in one row with victim memory. In this case the attacker has to restart the profiling phase.

After automatically switching to the exploitation phase we are able to monitor the exact timestamp of every keystroke in the address bar. We verified empirically that row hits can be measured on the found addresses after keystrokes by triggering keystrokes by hand. Figure 12 shows an access time trace for an address found in a DRAMA template attack, while typing in the Firefox address bar. For every key the user presses, a low access time is measured. We found this address after less than 2 seconds. Over 80 seconds we measured no false positive row hits and when pressing 40 keys we measured no false negatives. During this test the system was entirely idle apart from the attack and the user typing in Firefox. In a real attack, noise would introduce false negatives.

Comparison with cache template attacks. To compare DRAMA template attacks with cache template attacks, we performed two attacks on gedit. The first uses the result from a cache template attack in a DRAMA exploitation phase. The second is a modified cache template attack that uses the DRAMA side channel. Both attacks use shared memory to be able to compare them with cache template attacks. However, the DRAMA side-

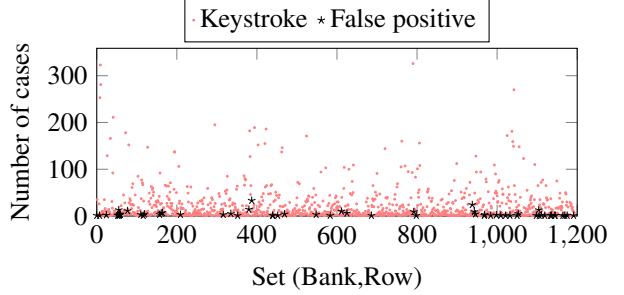


Figure 11: A DRAM template of the system memory with and without triggering keystrokes in the Firefox address bar. 1136 sets had row hits after a keystroke, 59 sets had false positive row hits (row hits without a keystroke), measured on our Ivy Bridge i5 test system.

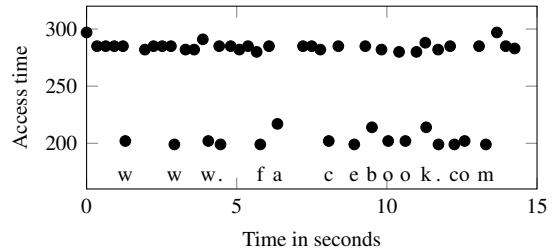


Figure 12: Exploitation phase on non-shared memory in a DRAMA template attack on our Ivy Bridge i5 test system. A low access time is measured when the user presses a key in the Firefox address bar. The typing gaps illustrate the low noise level.

channel attack takes no advantage of shared memory in any attack.

In the first attack on gedit, we target tab open and tab close events. In an experiment over 120 seconds we opened a new tab and closed the new tab, each 50 times. The exploitable address in the shared library was found in a cache template attack. We computed the physical address and thus bank and row of the exploitable address using privileged operating services. Then we allocated large arrays to obtain memory that maps to the same row (and bank). This allows us to perform an attack that has only minimal differences to a Flush+Reload attack.

During this attack, our spy tool detected 1 false positive row hit and 1 false negative row hit. Running stress $-m 1$ in parallel, which allocates and accesses large memory buffers, causes a high number of cache misses, but did not introduce a significant amount of noise. In this experiment the spy tool detected no false positive row hits and 4 false negative row hits. Running stress $-m 2$ in parallel (*i.e.*, the attacker's core is under stress) made any measurements impossible. While no false positive detections occurred, only 9 events were

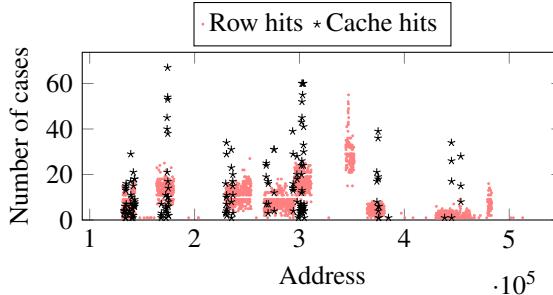


Figure 13: Comparison of a cache hits and row hits over the virtual memory where the gedit binary is mapped, measured on our Ivy Bridge i5 test system.

correctly detected. Thus, our attack is susceptible to noise especially if the attacker only gets a fraction of CPU time on its core.

In the second attack we compared the cache side channel and the DRAM side channel in a template attack on keystrokes in gedit. Figure 13 shows the number of cache hits and row hits over the virtual memory where the gedit binary is mapped. Row hits occur in spatial proximity to the cache hits and at shifted offsets due to the DRAM address mappings.

6.3 Comparison with state of the art

We now compare DRAMA side-channel attacks with same-CPU cache attacks such as Flush+Reload and Prime+Probe, as well as with cross-CPU cache attacks [11]. Our attack is the first to enable monitoring non-shared memory cross-CPU with a reasonably high spatial accuracy and a timing accuracy that is comparable to Flush+Reload. This allows the development of new attacks on programs using dynamically allocated or private memory.

The spatial accuracy of the DRAMA side-channel attack is significantly higher than that of a Prime+Probe attack, which also does not necessitate shared memory, and only slightly lower than that of a Flush+Reload attack in most cases. Our Ivy Bridge i5 system has 8 GB DRAM and a 3 MB L3 cache that is organized in 2 cache slices with each 2048 cache sets. Thus, in a Prime+Probe attack 32768 memory lines map to the same cache set, whereas in our DRAMA side-channel attack, on the same system, only 32 memory lines map to the same row. The spatial accuracy strongly depends on the system. On our Haswell-EP system only 8 memory lines map to the same row whereas still 32768 memory lines map to the same cache set. Thus, on the Haswell-EP system the advantage of DRAMA side-channel attacks over Prime+Probe is even more significant.

To allocate memory lines that are in the same row as victim memory lines, it is necessary to allocate significantly larger memory buffers than in a cache attack like Prime+Probe. This is a clear disadvantage of DRAMA side-channel attacks. However, DRAMA side-channel attacks have a very low probability of false positive row hit detections, whereas Prime+Probe is highly susceptible to noise. Due to this noise, monitoring singular events using Prime+Probe is extremely difficult.⁵

Irazoqui et al. [11] presented cache-based cross-CPU side-channel attacks. However, their work requires shared memory. Our approach works without shared memory. Not only does this allow cross-CPU attacks in highly restricted environments, it also allows to perform a new kind of cross-core attack within one system.

7 Improving attacks

In this section, we describe how the DRAM addressing functions can be used to improve the accuracy, efficiency, and success rate of existing attacks.

Flush+Reload. The first step when performing Flush+Reload attacks is to compute a cache-hit threshold, based on a histogram of cache hits and cache misses (memory accesses). However, as we have shown (cf. Figure 1) row hits have a slightly lower access time than row conflicts. To get the best performance in a Flush+Reload attack it is necessary to take row hits and conflicts into account. Otherwise, if a process accesses any memory location in the same row, a row hit will be misclassified as a cache hit. This introduces a significant amount of noise as the spatial accuracy of a cache hit is 64 bytes and the one of a row hit can be as low as 8 KB, depending on how actively the corresponding pages of the row are used. We found that even after a call to `sched_yield` a row hit is still observed in 2% of the cases on a Linux system that is mostly idle. In a Flush+Reload attack the victim computes in parallel and thus the probability then is even higher than 2%. This introduces a significant amount of noise especially for Flush+Reload attacks on low-frequency events. Thus, the accuracy of Flush+Reload attacks can be improved significantly taking row hits into account for the cache hit threshold computation.

Rowhammer. In a Rowhammer attack, an adversary tries to trigger bit flips in DRAM by provoking a high number of row switches. The success rate and efficiency of this attack benefit greatly from knowing the DRAM mapping, as we now demonstrate.

In order to cause row conflicts, one must alternately access addresses belonging to the same bank, but different row. The probability that 2 random addresses fulfill this criterion is 2^{-B} , where B is the total number of

bank-addressing bits (this includes all bits for channel, rank, etc.). For instance, with the dual-channel DDR4 configuration shown in Figure 4c this probability is only $2^{-6} = 1/64$. By hammering a larger set of addresses, the probability of having at least two targeting the same bank increases. However, so does the time in between row switches, thus the success rate decreases.

The most efficient way of performing the Rowhammer attack is *double-sided hammering*. Here, one tries to cause bit flips in row n by alternately accessing the adjacent rows $n - 1$ and $n + 1$, which are most likely also adjacent in physical memory. The most commonly referenced implementation of the Rowhammer attack, by Seaborn and Dullien [24], performs double-sided hammering by making assumptions on, e.g., the position of the row-index bits. If these are not met, then their implementation does not find any bit flips. Also, it needs to test multiple address combinations as it does not use knowledge of the DRAM addressing functions. We tested their implementation on a Skylake machine featuring G.SKILL F4-3200C16D-16GTZB DDR4 memory at the highest possible refresh interval, yet even after 4 days of nonstop hammering, we did not detect any bit flips.

By using the DRAM addressing functions we can immediately determine whether two addresses map to the same bank. Also, we can very efficiently search for pairs allowing double-sided hammering. After taking the reverse-engineered addressing functions into account, we successfully caused bit flips on the same Skylake setup within minutes. Running the same attack on a Crucial DDR4-2133 memory module running at the default refresh interval, we observed the first bit flip after 16 seconds and subsequently observed on average one bit flip every 12 seconds. Although the LPDDR4 standard includes *target row refresh* (TRR) as an optional countermeasure against the Rowhammer attack, the DDR4 standard does not. Still, some manufacturers include it in their products as a non-standard feature. For both DDR4 and LPDDR4, both the memory controller and the DRAM must support this feature in order to provide any protection. To the best of our knowledge, both our Haswell-EP test system and the Crucial DDR4-2133 memory module, with Micron DRAM chips, support TRR [10, 17]. However, we are still able to reproducibly trigger bit flips in this configuration.

8 Countermeasures

Defending against row buffer attacks is a difficult task. Making the corresponding DRAM operations constant time would introduce unacceptable performance degradation. However, as long as the timing difference exists and can be measured, the side channel cannot be closed.

Our attack implementations use the unprivileged `clflush` instruction in order to cause a DRAM access with every memory request. Thus, one countermeasure might be to restrict said operation. However, this requires architectural changes and an attacker can still use eviction as a replacement. The additional memory accesses caused by eviction could make our row-buffer covert channel impractical. However, other attacks such as the fully automated reverse engineering or our row-hit side-channel attack are still possible. Restricting the `rdtsc` instruction would also not prevent an attack as other timing sources can be used as replacement.

To prevent cross-VM attacks on multi-CPU cloud systems, the cloud provider could schedule each VM on a dedicated physical CPU and only allow access to CPU-local DRAM. This can be achieved by using a non-interleaved NUMA configuration and assigning pages to VMs carefully. This approach essentially splits a multi-CPU machine into independent single-CPU systems, which leads to a loss of many of its advantages.

Saltaformaggio et al. [21] presented a countermeasure to the memory bus-based covert channel of Wu et al.. It intercepts atomic instructions that are responsible for this covert channel, so that only cores belonging to the attacker’s VM are locked, instead of the whole machine. This countermeasure is not effective against our attacks as they do not rely on atomic instructions.

Finally, our attack could be detected due to the high number of cache misses. However, it is unclear whether it is possible to distinguish our attacks from non-malicious applications.

9 Conclusion

In this paper, we presented two methods to reverse engineer the mapping of physical memory addresses to DRAM channels, ranks, and banks. One uses physical probing of the memory bus, the other runs entirely in software and is fully automated. We ran our method on a wide range of architectures, including desktop, server, and mobile platforms.

Based on the reverse-engineered functions, we demonstrated DRAMA (DRAM addressing) attacks. This novel class of attacks exploits the DRAM row buffer that is a shared resource in single and multi-processor systems. This allows our attacks to work in the most restrictive environments, *i.e.*, across processors and without any shared memory. We built a covert channel with a capacity of 2 Mbps, which is three to four orders of magnitude faster than memory-bus-based channels in the same setting. We demonstrated a side-channel template attack automatically locating and monitoring memory accesses, *e.g.*, user input, server requests. This side-channel attack is as accurate as recent cache attacks like

Flush+Reload, while requiring no shared memory between the victim and the spy. Finally, we show how to use the reverse-engineered DRAM addressing functions to improve existing attacks, such as Flush+Reload and Rowhammer. Our work enables practical Rowhammer attacks on DDR4.

We emphasize the importance of reverse engineering microarchitectural components for security reasons. Before we reverse engineered the DRAM address mapping, the DRAM row buffer was transparent to operating system and software. Only by reverse engineering we made this shared resource visible and were able to identify it as a powerful side channel.

Acknowledgments

We would like to thank our anonymous reviewers as well as Anders Fogh, Moritz Lipp, and Mark Lanteigne for their valuable comments and suggestions.

Supported by the EU FP7 programme under GA No. 610436 (MATTHEW) and the Austrian Research Promotion Agency (FFG) under grant number 845579 (MEMSEC).

References

- [1] ADVANCED MICRO DEVICES. BIOS and Kernel Developer’s Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, 2013. URL: http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf.
- [2] BINGER, N., VAN DE POOL, J., SMART, N. P., AND YAROM, Y. “Ooh Aah... Just a Little Bit” : A small amount of side channel can go a long way. In *Proceedings of the 16th Workshop on Cryptographic Hardware and Embedded Systems (CHES’14)* (2014), pp. 75–92.
- [3] GRUSS, D., BIDNER, D., AND MANGARD, S. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS’15)* (2015).
- [4] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA’16* (2016).
- [5] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA’16* (2016).
- [6] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), USENIX Association.
- [7] HASSAN, M., KAUSHIK, A. M., AND PATEL, H. Reverse-engineering embedded memory controllers through latency-based analysis. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE* (2015), IEEE, pp. 297–306.
- [8] HUANG, R.-F., YANG, H.-Y., CHAO, M. C.-T., AND LIN, S.-C. Alternate hammering test for application-specific DRAMs and an industrial case study. In *Proceedings of the 49th Annual Design Automation Conference (DAC’12)* (2012), pp. 1012–1017.
- [9] INCI, M. S., GULMEZOGLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *Cryptology ePrint Archive, Report 2015/898* (2015), 1–15.
- [10] INTEL CORPORATION. *Intel® Xeon® Processor E5 v3 Product Family – Processor Specification Update*. No. 330785-009US. Aug. 2015.
- [11] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cross processor cache attacks. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security* (2016), ASIA CCS ’16, ACM.
- [12] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Wait a minute! A fast, Cross-VM attack on AES. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID’14)* (2014).
- [13] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture – ISCA* (2014), pp. 361–372.
- [14] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P’15)* (2015).
- [15] MAURICE, C., LE SCOURNEC, N., NEUMANN, C., HEEN, O., AND FRANCILLON, A. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID’15)* (2015).
- [16] MAURICE, C., NEUMANN, C., HEEN, O., AND FRANCILLON, A. C5: Cross-Cores Cache Covert Channel. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA’15)* (July 2015).
- [17] MICRON. DDR4 SDRAM. https://www.micron.com/~media/documents/products/data-sheet/dram/ddr4/4gb_ddr4_sdram.pdf, 2014. Retrieved on February 17, 2016.
- [18] PARK, K., BAEG, S., WEN, S., AND WONG, R. Active-Precharge Hammering on a Row Induced Failure in DDR3 SDRAMs under 3x nm Technology. In *Proceedings of the 2014 IEEE International Integrated Reliability Workshop Final Report (IIRW’14)* (2014), pp. 82–85.
- [19] PERCIVAL, C. Cache Missing for Fun and Profit, 2005. URL: <http://daemonology.net/hyperthreading-considered-harmful/>.
- [20] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM Conference on Computer and Communications Security – CCS* (2009), ACM, pp. 199–212.
- [21] SALTAFORMAGGIO, B., XU, D., AND ZHANG, X. BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels. In *Proceedings of the 6th European Workshop on Systems Security (EuroSec’13)* (2013).
- [22] SEABORN, M. Exploiting the DRAM rowhammer bug to gain kernel privileges. <http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html>, March 2015. Retrieved on June 26, 2015.
- [23] SEABORN, M. How physical addresses map to rows and banks in DRAM. <http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html>, May 2015. Retrieved on July 20, 2015.

- [24] SEABORN, M., AND DULLIEN, T. Test DRAM for bit flips caused by the rowhammer problem. <https://github.com/google/rowhammer-test>, 2015. Retrieved on July 27, 2015.
- [25] WU, Z., XU, Z., AND WANG, H. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. *IEEE/ACM Transactions on Networking* (2014).
- [26] XIAO, J., XU, Z., HUANG, H., AND WANG, H. Security implications of memory deduplication in a virtualized environment. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)* (June 2013), Ieee, pp. 1–12.
- [27] XIAO, Y., ZHANG, X., ZHANG, Y., AND TEODORESCU, M.-R. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *25th USENIX Security Symposium* (2016).
- [28] YAROM, Y., AND FALKNER, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23th USENIX Security Symposium* (2014).
- [29] YAROM, Y., GE, Q., LIU, F., LEE, R. B., AND HEISER, G. Mapping the Intel Last-Level Cache. *Cryptology ePrint Archive, Report 2015/905* (2015), 1–12.
- [30] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS'12)* (2012).

An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries

Dennis Andriesse^{†‡}, Xi Chen^{†‡}, Victor van der Veen^{†‡}, Asia Slowinska[‡], and Herbert Bos^{†‡}

[†]{d.a.andriesse,x.chen,v.vander.veen,h.j.bos}@vu.nl

Computer Science Institute, Vrije Universiteit Amsterdam

[‡]Amsterdam Department of Informatics

[‡]asia@lastline.com

Lastline, Inc.

Abstract

It is well-known that static disassembly is an unsolved problem, but how much of a problem is it in real software—for instance, for binary protection schemes? This work studies the accuracy of nine state-of-the-art disassemblers on 981 real-world compiler-generated binaries with a wide variety of properties. In contrast, prior work focuses on isolated corner cases; we show that this has led to a widespread and overly pessimistic view on the prevalence of complex constructs like inline data and overlapping code, leading reviewers and researchers to underestimate the potential of binary-based research. On the other hand, some constructs, such as function boundaries, are much harder to recover accurately than is reflected in the literature, which rarely discusses much needed error handling for these primitives. We study 30 papers recently published in six major security venues, and reveal a mismatch between expectations in the literature, and the actual capabilities of modern disassemblers. Our findings help improve future research by eliminating this mismatch.

1 Introduction

The capabilities and limitations of disassembly are not always clearly defined or understood, making it difficult for researchers and reviewers to judge the practical feasibility of techniques based on it. At the same time, disassembly is the backbone of research in static binary instrumentation [5, 19, 32], binary code lifting to LLVM IR (for reoptimization or analysis) [38], binary-level vulnerability search [27], and binary-level anti-exploitation systems [1, 8, 29, 46]. Disassembly is thus crucial for analyzing or securing untrusted or proprietary binaries, where source code is simply not available.

The accuracy of disassembly strongly depends on the type of binary under analysis. In the most general case, the disassembler can make very few assumptions on the structure of a binary—high-level concepts like functions and loops have no real significance at the binary level [3].

Moreover, the binary may contain complex constructs, such as overlapping or self-modifying code, or inline data in executable regions. This is especially true for obfuscated binaries, making disassembly of such binaries extremely challenging. Disassembly in general is undecidable [43]. On the other hand, one might expect that compilers emit code with more predictable properties, containing a limited set of patterns that the disassembler may try to identify.

Whether this is true is not well recognized, leading to a wide range of views on disassembly. These vary from the stance that disassembly of benign binaries is a solved problem [48], to the stance that complex cases are rampant [23]. It is unclear which view is justified in a given situation. The aim of our work is thus to study binary disassembly in a realistic setting, and more clearly delineate the capabilities of modern disassemblers.

It is clear from prior work that obfuscated code may complicate disassembly in a myriad of ways [18, 21]. We therefore limit our study to non-obfuscated binaries compiled on modern x86 and x64 platforms (the most common in binary analysis and security research). Specifically, we focus on binaries generated with the popular `gcc`, `clang` and Visual Studio compilers. We explore a wide variety of 981 realistic binaries, including stripped, optimized, statically linked, and link-time optimized binaries, as well as library code that includes handcrafted assembly. We disassemble these binaries using nine state-of-the-art research and industry disassemblers, studying their ability to recover all disassembly primitives commonly used in the literature: instructions, function start addresses, function signatures, Control Flow Graphs (CFG) and callgraphs. In contrast, prior studies focus strongly on complex corner cases in isolation [23, 25]. Our results show that such cases are exceedingly rare, even in optimized code, and that focusing on them leads to an overly pessimistic view on disassembly.

We show that many disassembly primitives can be recovered with better accuracy than previously thought. For

instance, instruction accuracy often approaches 100%, even using linear disassembly. On the other hand, we also identify some primitives which are more difficult to recover—most notably, function start information.

To facilitate a better match between the capabilities of disassemblers and the expectations in the literature, we comprehensively study all binary-based papers published in six major security conferences in the last three years. Ironically, this study shows a focus in the literature on rare complex constructs, while little attention is devoted to error handling for primitives that really are prone to inaccuracies. For instance, only 25% of Windows-targeted papers that rely on function information discuss potential inaccuracies, even though the accuracy of function detection regularly drops to 80% or less. Moreover, less than half of all papers implement mechanisms to deal with inaccuracies, even though in most cases errors can lead to malignant failures like crashes.

Contributions & Outline

The contributions of our work are threefold.

- (1) We study disassembly on 981 full-scale compiler-generated binaries, to clearly define the true capabilities of modern disassemblers (Section 3) and the implications on binary-based research (Section 4).
- (2) Our results allow researchers and reviewers to accurately judge future binary-based research—a task currently complicated by the myriad of differing opinions on the subject. To this end, we release all our raw results and ground truth for use in future evaluations of binary-based research¹.
- (3) We analyze the quality of all recent binary-based work published in six major security venues by comparing our results to the requirements and assumptions of this work (Section 5). This shows where disassembler capabilities and the literature are mismatched, and how this mismatch can be resolved moving forward (Section 6).

2 Evaluating Real-World Disassembly

This section outlines our disassembly evaluation approach. We discuss our results, and the implications on binary-based research, in Sections 3–4. Sections 5–6 discuss how closely expectations in the literature match our results.

2.1 Binary Test Suite

We focus our analysis on non-obfuscated x86 and x64 binaries generated with modern compilers. Our experiments are based on Linux (ELF) and Windows (PE) binaries, generated with the popular `gcc` v5.1.1, `clang` v3.7.0 and

¹<https://www.vusec.net/projects/disassembly/>

Visual Studio 2015 compilers—the most recent versions at the time of writing. The x86/x64 instruction set is the most common target in binary-based research. Moreover, x86/x64 is a variable-length instruction set, allowing unique constructs such as overlapping and “misaligned” instructions which can be difficult to disassemble. We exclude obfuscated binaries, as there is no doubt that they can wreak havoc on disassembler performance and we hardly need confirm this in our experiments.

We base our disassembly experiments on a test suite composed of the SPEC CPU2006 C and C++ benchmarks, the widely used and highly optimized `glibc-2.22` library, and a set of popular server applications consisting of `nginx` v1.8.0, `lighttpd` v1.4.39, `opensshd` v7.1p2, `vsftpd` v3.0.3 and `exim` v4.86. This test suite has several properties which make it representative: (1) It contains a wide variety of realistic C and C++ binaries, ranging from very small to large; (2) These correspond to binaries used in evaluations of other work, making it easier to relate our results to the literature; (3) The tests include highly optimized library code, containing handwritten assembly and complex corner cases which regular applications do not; (4) SPEC CPU2006 compiles on both Linux and Windows, allowing a fair comparison of results between `gcc`, `clang`, and Visual Studio.

To study the impact of compiler options on disassembly, we compile the SPEC CPU2006 part of our test suite multiple times with a variety of popular configurations. Specifically: (1) Optimization levels 00, 01, 02 and 03 for `gcc`, `clang` and Visual Studio; (2) Optimization for size (0s) on `gcc` and `clang`; (3) Static linking and link-time optimization (-fLTO) on 64-bit `gcc`; (4) Stripped binaries, as well as binaries with symbols. We compile the servers for both x86 and x64 with `gcc` and `clang`, leaving all remaining settings at the Makefile defaults. Finally, we compile `glibc-2.22` with 64-bit `gcc`, to which it is specifically tailored. In total, our test suite contains 981 binaries and shared objects.

2.2 Disassembly Primitives

We test all five common disassembly primitives used in the literature (see Section 5). Some of these go well beyond basic instruction recovery, and are only supported by a subset of the disassemblers we test.

- (1) *Instructions*: The pure assembly-level instructions.
- (2) *Function starts*: Start addresses of the functions originally defined in the source code.
- (3) *Function signatures*: Parameter lists for functions found by the disassembler.
- (4) *Control Flow Graph (CFG) accuracy*: The soundness and completeness of the CFG digraphs $G_{cfg} = (V_{bb}, E_{cfg})$, which describe how control flow edges $E_{cfg} \subseteq V_{bb} \times V_{bb}$ connect the basic blocks V_{bb} . In practice, dis-

assemblers deviate from the traditional CFG; typically by omitting indirect edges, and sometimes by defining a global CFG rather than per-function CFGs. Therefore, we define the *Interprocedural CFG (ICFG)*: the union of all function-level CFGs, connected through interprocedural call and jump edges. This allows us to abstract from the disassemblers’ varying CFG definitions, by focusing our measurement on the coverage of basic blocks in the ICFG. We pay special attention to hard-to-resolve basic blocks, such as the heads of address-taken functions and switch/case blocks reached via jump tables.

(5) *Callgraph accuracy*: The correctness of the digraph $G = (V_{cs} \cup V_f, E_{call})$ linking the set V_{cs} of call sites to the function starts V_f through call edges $E_{call} \subseteq V_{cs} \times V_f$. Similarly to the CFG, disassemblers deviate from the traditional callgraph definition by including only direct call edges. In our experiments, we therefore measure the completeness of this *direct callgraph*, considering indirect calls and tailcalls separately in our complex case analysis.

2.3 Complex Constructs

We also study the prevalence in real-world binaries of complex corner cases which are often cited as particularly harmful to disassembly [5, 23, 34].

(1) *Overlapping/shared basic blocks*: Basic blocks may be shared between different functions, hindering disassemblers from properly separating these functions.

(2) *Overlapping instructions*: Since x86/x64 uses variable-length instructions without any enforced memory alignment, jumps can target any offset within a multi-byte instruction. This allows the same code bytes to be interpreted as multiple overlapping instructions, some of which may be missed by disassemblers.

(3) *Inline data and jump tables*: Data bytes may be mixed in with instructions in a code section. Examples of potential inline data include jump tables or local constants. Such data can cause false positive instructions, and can desynchronize the instruction stream if the last few data bytes are mistakenly interpreted as the start of a multi-byte instruction. Disassembly then continues parsing this instruction into the actual code bytes, losing track of the instruction stream alignment.

(4) *Switches/case blocks*: Switches are a challenge for basic block discovery, because the switch case blocks are typically indirect jump targets (encoded in jump tables).

(5) *Alignment bytes*: Some code (i.e., `nop`) or data bytes may have no semantic meaning, serving only to align other code for optimization of memory accesses. Alignment bytes may cause desynchronization if they do not encode valid instructions.

(6) *Multi-entry functions*: Functions may have multiple basic blocks used as entry points, which can complicate function start recognition.

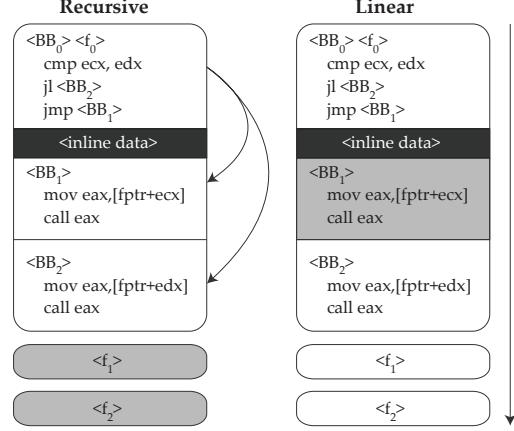


Figure 1: Disassembly methods. Arrows show disassembly flow. Gray blocks show missed or corrupted code.

(7) *Tail calls*: In this common optimization, a function ends not with a return, but with a jump to another function. This makes it more difficult for disassemblers to detect where the optimized function ends.

2.4 Disassembly & Testing Environment

We conducted all disassembly experiments on an Intel Core i5 4300U machine with 8GB of RAM, running Ubuntu 15.04 with kernel 3.19.0-47. We compiled our `gcc` and `clang` test cases on this same machine. The Visual Studio binaries were compiled on an Intel Core i7 3770 machine with 8GB of RAM, running Windows 10.

We tested nine popular industry and research disassemblers: *IDA Pro v6.7*, *Hopper v3.11.5*, *Dyninst v9.1.0* [5], *BAP v0.9.9* [7], *ByteWeight v0.9.9* [4], *Jakstab v0.8.4* [17], *angr v4.6.1.4* [36], *PSI v1.1* [47] (the successor of BinCFI [48]), and *objdump v2.22*. ByteWeight yields only function starts, while Dyninst and PSI support only ELF binaries (for Dyninst, this is due to our Linux testing environment). Jakstab supports only x86 PE binaries. We omit angr results for x86, as angr is optimized for x64. PSI is based on objdump, with added error correction. Section 3 shows that PSI (and all linear disassemblers) perform equivalently to objdump; therefore, we group these under the name *linear disassembly*.

All others are recursive descent disassemblers, illustrated in Figure 1. These follow control flow to avoid desynchronization by inline data, and to discover complex cases like overlapping instructions. In contrast, linear disassemblers like objdump simply decode all code bytes consecutively, and may be confused by inline data, possibly causing garbled code like BB_1 in the figure. Recursive disassemblers avoid this problem, but may miss indirect control flow targets, such as f_1 and f_2 in the figure.

2.5 Ground Truth

Our disassembly experiments require precise ground truth on instructions, basic blocks and function starts, call sites, function signatures and switch/case addresses. This information is normally only available at the source level. Clearly, we cannot obtain our ground truth from any disassembler, as this would bias our experiments.

We base our ELF ground truth on information collected by an LLVM analysis pass, and on DWARF v3 debugging information. Specifically, we use LLVM to collect source-level information, such as the source lines belonging to functions and switch statements. We then compile our test binaries with DWARF information, and link the source-level line numbers to the binary-level addresses using the DWARF line number table. We also use DWARF information on function parameters for our function signature analysis. We strip the DWARF information from the binaries before our disassembly experiments.

The line number table provides a full mapping of source lines to binary, but not all instructions correspond directly to a source line. To find these instructions, we use *Capstone v3.0.4* to start a conservative linear disassembly sweep from each known instruction address, stopping at control flow instructions unless we can guarantee the validity of their destination and fall-through addresses. For instance, the target of a direct unconditional jump instruction can be guaranteed, while its fall-through block cannot (as it might contain inline data).

This approach yields ground truth for over 98% of code bytes in the tested binaries. We manually analyze the remaining bytes, which are typically alignment code unreachable by control flow. The result is a ground truth file for each binary test case, that specifies the type of each code byte, as well as instruction and function starts, switch/case addresses, and function signatures.

We use a similar method for the Windows PE tests, but based on information from PDB (Program Database) files produced by Visual Studio instead of DWARF. This produces files analogous to our ELF ground truth format.

We release all our ground truth files and our test suite, to aid in future evaluations of binary-based research and disassembly.

3 Disassembly Results

This section describes the results of our disassembly experiments, using the methodology outlined in Section 2. We first discuss application binaries (SPEC and servers), followed by a separate discussion on highly optimized libraries. Finally, we discuss the impact of static linking and link-time optimization. We release all our raw results, and present aggregated results here for space reasons.

3.1 Application Binaries

This section presents disassembly results for application code. We discuss accuracy results for all primitives, and also analyze the prevalence of complex cases.

3.1.1 SPEC CPU2006 Results

Figures 2a–2e show the accuracy for the SPEC CPU2006 C and C++ benchmarks of the recovered instructions, function starts, function signatures, CFGs and callgraphs, respectively. We show the percentage of correctly recovered (true positive) primitives for each tested compiler at optimization levels 00–03. Note that the legend in Figure 2a applies to Figures 2a–2e. All lines are geometric mean results (simply referred to as “mean” from this point); arithmetic means and standard deviations are discussed in the text where they differ significantly. We show separate results for the C and C++ benchmarks, to expose variations in disassembly accuracy that may result from different code patterns.

Some disassemblers support only a subset of the tested primitives. For instance, linear disassembly provides only instructions, and IDA Pro is the only tested disassembler that provides function signatures. Moreover, some disassemblers only support a subset of the tested binary types, and are therefore only shown in the plots where they are applicable. For clarity, the graphs only show results for stripped binaries; our tests with standard symbols (not DWARF information) are discussed in the text.

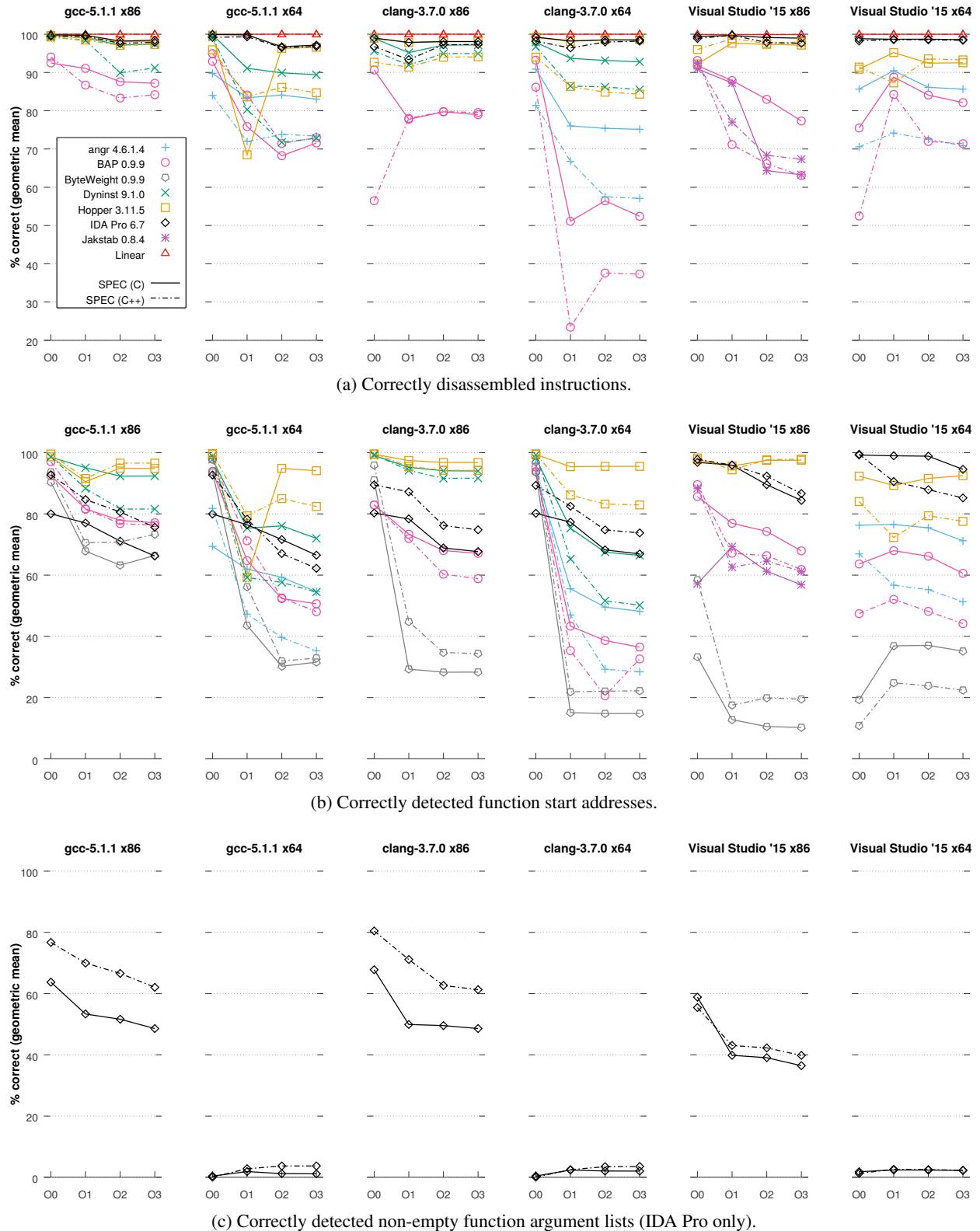
3.1.1.1 Instruction boundaries

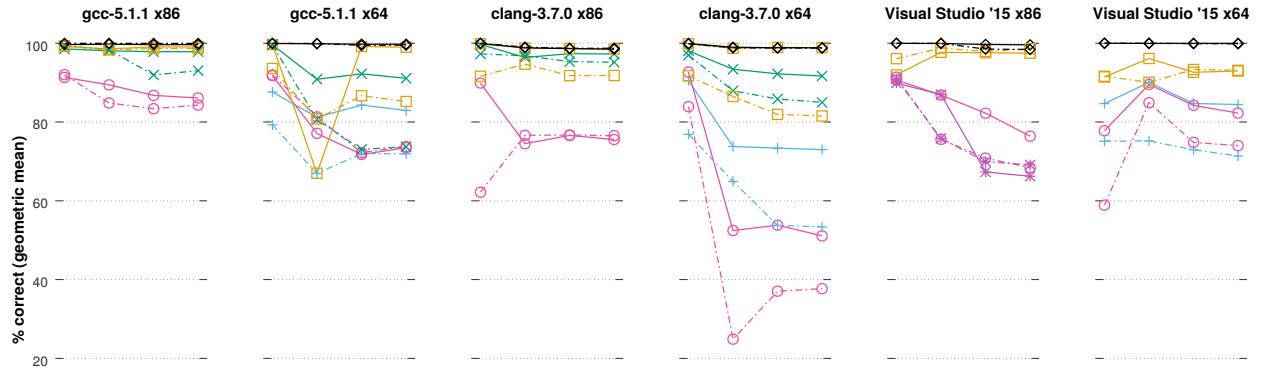
Figure 2a shows the percentage of correctly recovered instructions. Interestingly, linear disassembly consistently outperforms all other disassemblers, finding 100% of the instructions for `gcc` and `clang` binaries (without false positives), and 99.92% in the worst case for Visual Studio.

Linear disassembly. The perfect accuracy for linear disassembly with `gcc` and `clang` owes to the fact that these compilers never produce inline data, not even for jump tables. Instead, jump tables and other data are placed in the `.rodata` section.

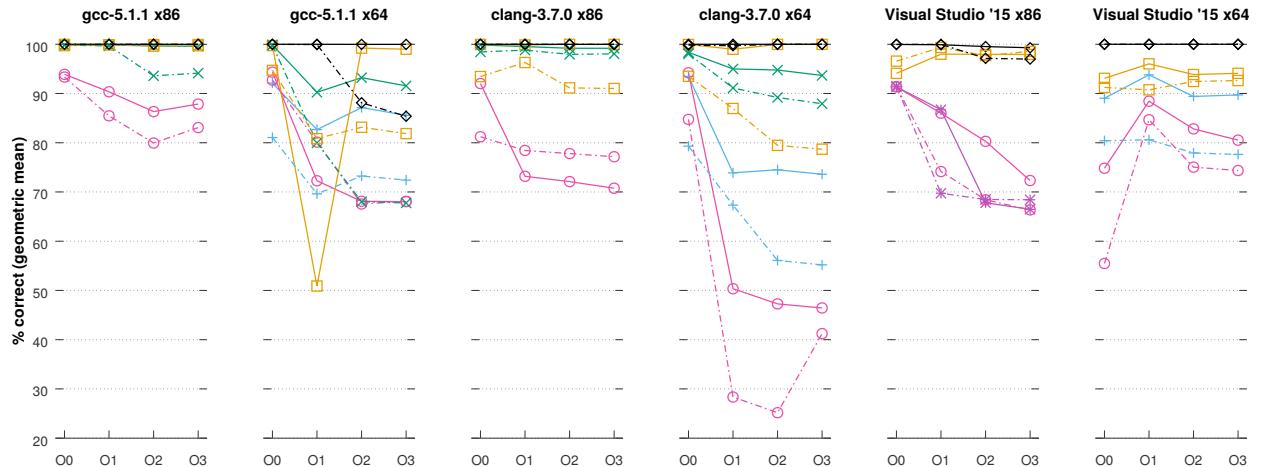
Visual Studio *does* produce inline data, typically jump tables. This leads to some false positives with linear disassembly (data treated as code), amounting to a worst-case mean of 989 false positive instructions (0.56% of the disassembled code) for the x86 C++ tests at 03. The number of missed instructions (false negatives, due to desynchronization) is much lower, at a worst-case mean of 0.09%. This is because x86/x64 disassembly automatically resynchronizes within two or three instructions [21].

Figure 2: Disassembly results. The legend in Figure 2a applies to Figures 2a–2e. Section 2.4 describes which platforms are supported by each tested disassembler.





(d) Correct and complete basic blocks for the ICFG.



(e) Correctly resolved direct function calls (indirect calls discussed separately).

Recursive disassembly. The most accurate recursive disassembler in terms of instruction recovery is IDA Pro 6.7, which closely follows linear disassembly with an instruction coverage exceeding 99% at optimization levels 00 and 01, dropping to a worst case mean of 96% for higher optimization levels. The majority of missed instructions at higher optimization levels are alignment code for functions and basic blocks, which is quite common in optimized binaries. It consists of various (long) nop instructions for `gcc` and `clang`, and of `int 3` instructions for `Visual Studio`, and accounts for up to 3% of all code at 02 and 03. Missing these instructions is not harmful to common binary analysis operations, such as binary instrumentation, manual analysis or decompilation.

False positives in IDA Pro are less prevalent than in linear disassembly. On `gcc` and `clang`, they are extremely rare, amounting to 14 false positives in the worst test case, with a mean of 0. `Visual Studio` binaries produce more false positives, peaking at 0.16% of all recovered instructions. Overall, linear disassembly provides the most

complete instruction listing, but at a relatively high false positive rate for `Visual Studio`. IDA Pro finds only slightly fewer instructions, with significantly fewer false positives. These numbers were no better for binaries with symbols.

Dyninst and Hopper achieve best case accuracy comparable to IDA, but not quite as consistently. Some disassemblers, notably BAP, appear to be optimized for `gcc`, and show large performance drops when used on `clang`. The BAP authors informed us that BAP's results depend strongly on the disassembly starting points (i.e., function starts), provided by ByteWeight. We used the default ELF and PE signature files shipped with ByteWeight v0.9.9. Our angr results are based on the CFGFast analysis recommended to us by the angr authors.

Overall, IDA Pro, Hopper, Dyninst and linear disassembly show arithmetic mean results which are extremely close to the geometric means, exhibiting standard deviations below 1%. Other disassemblers have larger standard deviations, typically around 15%, with outliers up to 36% (for BAP on `clang` x86, as visible in Figure 2a).

```

6caf10 <ix86_fp.compare_mode>:
6caf10: mov 0x3f0dde(%rip),%eax
6caf16: and $0x10,%eax
6caf19: cmp $0x1,%eax
6caf1c: sbb %eax,%eax
6caf1e: add $0x3a,%eax
6caf21: retq

```

Listing 1: False negative indirectly called function for IDA Pro in gcc, compiled with gcc at 03 for x64 ELF.

```

480970 <autohelperowl_defendpat156>:
480970: push %rbp
480971: push %r15
480973: push %r14
480975: push %rbx
480976: push %rax

```

Listing 2: False positive function (shaded) for Dyninst, due to misapplied prologue signature, gobmk compiled with clang at 01 for x64 ELF.

C versus C++. Accuracy between C and C++ differs most in the lower scoring disassemblers, but the difference largely disappears in the best performing disassemblers. The largest relative difference appears for clang.

3.1.1.2 Function starts

The results for function start detection are far more diffuse than those for instruction recovery. Consider Figure 2b, which shows the mean percentage of correctly recovered function start addresses. No one disassembler consistently dominates these results, though Hopper is at the upper end of the spectrum for most compiler configurations in terms of true positives. Dyninst also provides high true positive rates, though not as consistently as Hopper. However, as shown in Figure 3, both Hopper and Dyninst suffer from high false positive rates, with worst case mean false positive rates of 28% and 19%, respectively. IDA Pro provides lower false positive rates of under 5% in most cases (except for x86 Visual Studio, where it peaks at 20%). However, its true positive rate is substantially lower than those of Hopper and Dyninst, regularly missing 20% or more of functions even at low optimization levels. As with instruction recovery, the results for BAP and ByteWeight depend heavily on the compiler configuration, ranging from over 90% accuracy on gcc x86 at 00, to under 20% on clang x64.

Even for the best performing disassemblers, function start identification is far more challenging than instruction recovery. Accuracy drops particularly as the optimization level increases, repeatedly falling from close to 99% true positives at 00, to only 82% at 03, and worsened by high false positive rates. For IDA Pro, the worst case mean true positive rate is even lower, falling to 62% for C++ on x64 gcc at 03. Moreover, the standard deviation increases to over 15% even for IDA Pro.

```

8060985: pop %ebx
8060986: pop %esi
8060987: ret
8060988: nop
8060989: lea 0x(%esi,%eiz,1),%esi

```

Listing 3: False positive function (shaded) for Dyninst, due to code misinterpreted as epilogue, sphinx compiled with gcc at 02 for x86 ELF.

```

46b990 <Perl_pp_enterloop>:
[...]
46ba02: ja 46bb50 <Perl_pp_enterloop+0x1c0>
46ba08: mov %rsi,%rdi
46ba0b: shl %cl,%rdi
46ba0e: mov %rdi,%rcx
46ba11: and $0x46,%ecx
46ba14: je 46bb50 <Perl_pp_enterloop+0x1c0>
[...]
46bb47: pop %r12
46bb49: retq
46bb4a: nopw 0x0(%rax,%rax,1)
46bb50: sub $0x90,%rax

```

Listing 4: False positive function (shaded) for Dyninst, due to code misinterpreted as epilogue, perlbench compiled with gcc at 03 for x64 ELF.

False negatives. The vast majority of false negatives is caused by indirectly called or tailcalled functions (reached by a `jmp` instead of a `call`), as shown in Listing 1. This explains why the true positive rate drops steeply at high optimization levels, where tail calls and functions lacking standard prologues are common (see Section 3.1.3). Symbols, if available, help greatly in improving accuracy. They are used especially effectively by IDA Pro, which consistently yields over 99% true positives for binaries with symbols, even at higher optimization levels.

False positives. Several factors contribute to the false positive rate. We analyzed a random sample of 50 false positives for Dyninst, Hopper and IDA Pro, the three best performing disassemblers in function detection.

For Dyninst, false positives are mainly due to erroneously applied signatures for function prologues and epilogues. As an example, Listing 2 shows a false positive in Dyninst due to a misidentified prologue: Dyninst scans for the `push %r15` instruction (as well as several other prologue signatures), missing preceding instructions in the function. We observe similar cases for function epilogues. For instance, as shown in Listings 3 and 4, Dyninst assumes a new function following a `ret`; `nop` instruction sequence. This is not always correct: as shown in the examples, the same code pattern can result from a multi-exit function with padding between basic blocks. Note that both examples could be handled correctly by control flow and semantics-aware disassemblers. In Listing 4, there are intraprocedural jumps towards the basic block at `0x46bb50`, showing that it is not a new function.

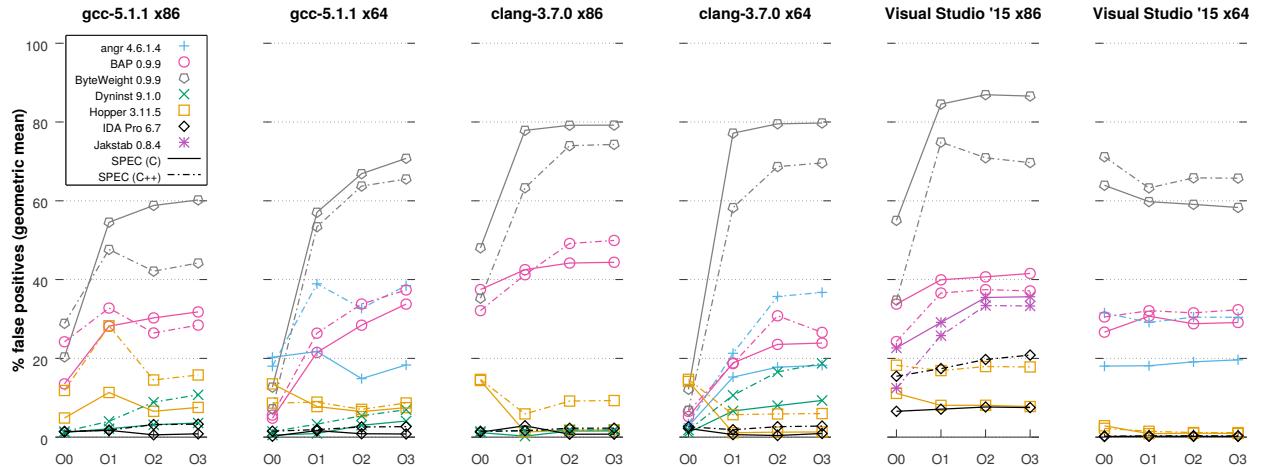


Figure 3: False positives for function start detection (percentage of total detected functions).

```

42cec3: movss %xmm0,-0x340(%rbp)
42ceb2: jmpq 42cf8 <P7PriorifyTransitionVector+0x622>
42ced0: mov -0x344(%rbp),%eax

```

Listing 5: False positive function (shaded) for Hopper, due to misclassified switch case block, hmmer compiled with gcc at 00 for x64 ELF.

The false positive in Listing 3 is in effect a `nop` instruction, emitted for padding by `gcc` on x86.

All false positives we sampled for Hopper are located directly after padding code, or after a direct `jmp` (without a fallthrough edge), and are not directly reached by other instructions. An example is shown in Listing 5. Since these instructions are never reached directly, Hopper assumes that they represent function starts. This is not always correct; for instance, the same pattern frequently results from case blocks belonging to switch statements, as seen in Listing 5.

Similarly, the majority of false positives for IDA Pro is also caused by unreachable code assumed to be a new function. However, these cases are far less common in IDA Pro than in Hopper, as IDA Pro more accurately resolves difficult control flow constructs such as switches. Interestingly, the false positive rate for IDA Pro drops to a mean of under 0.3% for x64 Visual Studio 2015. This is because 64-bit Visual Studio uses just one well-defined calling convention, while other compilers use a variety [22].

3.1.1.3 Function signatures

Of the tested disassemblers, only IDA Pro supports function signature analysis. Figure 2c shows the percentage of non-empty function argument lists where IDA Pro correctly identified the number of arguments. We focus on

non-empty argument lists because IDA Pro defaults to an empty list, skewing our results if counted as correct.

Argument recovery is far more accurate on x86 code, where parameters are typically passed on the stack, than it is on the register-oriented x64 architecture. For x86 code generated by `gcc` and `clang`, IDA Pro correctly identifies between 64% and 81% of the argument lists on non-optimized binaries, dropping to 48% in the worst case at 03. Results for Visual Studio are slightly worse, ranging from 36% worst case to 59% in the best case. As for function starts, the standard deviation is just over 15%. On x64 code, IDA Pro recovers almost none of the argument lists, with accuracy between 0.38% and 1.87%.

Performance is significantly better for binaries with symbols, even on x64, but only for C++ code. For instance, IDA Pro's accuracy for `gcc` x64 increases to a mean of 44% for C++, peaking at 75% correct argument lists. This is because IDA Pro parses mangled function names that occur in C++ symbols, which encode signature information in the function name.

3.1.1.4 Control Flow Graph accuracy

Figure 2d presents the accuracy of basic blocks in the ICFG, the union of all function-level CFGs. We found these results to be representative of the per-function CFG accuracy. The accuracy of the ICFG is strongly correlated with instruction discovery; indeed, recursive disassemblers typically find instructions through the process of expanding the ICFG itself. Thus, the disassemblers that perform well in instruction recovery also perform well in CFG construction. For some disassemblers, such as IDA Pro, the basic block true positive rate at high optimization levels even exceeds the raw instruction recovery results (Figure 2a). This is because for the ICFG, we did not count missing `nop` instructions as false negatives.

IDA Pro consistently achieves a basic block recovery rate of between 98–100%, even at high optimization levels. Even at moderate optimization levels, the results for Hopper and Dyninst are considerably less complete, regularly dropping to 90% or less. For the remaining disassemblers, basic block recovery rates of 75% or less are typical.

All disassemblers except IDA Pro show a considerable drop in accuracy on `gcc` and `clang` for x64, compared to the x86 results. This is strongly correlated with the diminishing instruction and function detection results for these disassembler/architecture combinations (see Figures 2a–2b). This implies that when functions are missed, these disassemblers also fail to recover the instructions and basic blocks contained in the missed functions. In contrast, IDA Pro disassembles instructions even when it cannot attribute them to any function. The difference between x86/x64 and C/C++ results is less pronounced for Visual Studio binaries than for `gcc`/`clang`.

3.1.1.5 Callgraph accuracy

Like ICFG accuracy, callgraph accuracy depends strongly on the completeness of the underlying instruction analysis. As mentioned, the callgraphs returned by the tested disassemblers contain only the direct call edges, and do not deal with address-taken functions. For this reason, Figure 2e presents results for the direct component of the callgraph only. We study the impact of indirect calls on function identification accuracy in our complex case analysis instead (Section 3.1.3). The direct callgraph results in Figure 2e again show IDA Pro to be the most accurate at a consistent 99% function call resolve rate (linking function call edges to function starts), in most cases followed closely by Dyninst and Hopper. This illustrates that the lower accuracy for function starts (Figure 2b) is mainly due to indirectly called functions (such as those called via function pointers or in tail call optimizations).

3.1.2 Server Results

Table 1 shows disassembly results for the servers from our test suite. For space reasons, and because the relative accuracy of the disassemblers is the same as for SPEC, we only show results for IDA Pro, the best overall disassembler. All other results are available externally, as mentioned at the start of Section 3. We compiled all servers for both x86 and x64 with `gcc` and `clang`, using their default Makefile optimization levels.

The server tests confirm that the SPEC results from Section 3.1.1 are representative; all results lie well within the established bounds. As with SPEC, linear disassembly achieved 100% correctness. The `nginx` results warrant closer inspection; given its optimization level 01, the

	x86					x64				
	Instructions	Functions	Signatures	ICFG	Callgraph	Instructions	Functions	Signatures	ICFG	Callgraph
gcc-5.1.1										
<code>nginx</code>	99.9	65.5	49.6	100	100	99.9	59.2	0.9	99.9	100
<code>lighttpd</code>	99.9	99.5	85.9	99.9	100	99.9	99.5	0.0	99.9	100
<code>vsftpd</code>	95.4	93.4	73.6	95.9	99.5	93.0	92.5	4.3	99.9	100
<code>opensshd</code>	99.9	86.2	74.9	100	100	99.9	86.2	0.0	100	100
<code>exim</code>	99.9	90.1	58.2	99.9	100	99.9	89.9	4.5	99.9	100
clang-3.7.0										
<code>nginx</code>	98.5	57.5	44.0	99.5	100	98.6	53.0	0.7	99.4	100
<code>lighttpd</code>	98.7	99.5	87.9	99.9	100	99.0	99.5	0.0	99.9	100
<code>vsftpd</code>	96.8	93.3	72.9	99.8	100	97.0	92.0	6.6	99.5	99.9
<code>opensshd</code>	98.9	86.5	78.1	100	100	99.2	86.3	0.0	100	100
<code>exim</code>	99.0	82.7	54.6	99.3	100	99.1	81.7	5.4	99.4	100

Table 1: IDA Pro 6.7 disassembly results for server tests (% correct, per test case).

function start and argument information is on the low end of the accuracy spectrum. Closer analysis shows that this results from extensive use in `nginx` of indirect calls through function pointers; Section 3.1.1 shows that this negatively affects function information. Indeed, for all tested servers, the accuracy of function start detection is inversely proportional to the ratio of address-taken functions to the total number of instructions. This shows that coding style can carry through the compilation process to have a strong effect on disassembler performance.

3.1.3 Prevalence of Complex Constructs

Figure 4 shows the prevalence of complex constructs in SPEC CPU2006, which pose special disassembly challenges. We also analyzed these constructs in the server binaries, finding no significantly different results.

We did not encounter any overlapping or shared basic blocks in either the SPEC or server tests on any compiler. This is surprising, as these constructs are frequently cited in the literature [5, 17, 23]. Closer inspection showed that all the cited cases of overlapping blocks are due to constructs which we classify more specifically, namely overlapping instructions and multi-entry functions. These constructs are exceedingly rare, and occur almost exclusively in library code (discussed in Section 3.2.2). This finding fits with the examples seen in the literature, which all stem from library code, most commonly `glibc`.

No overlapping instructions occur in Linux application code, and only a handful in Windows code (with a mean of zero, and a maximum of 3 and 10 instructions for x86 and x64 Visual Studio, respectively). Multi-entry functions are somewhat more common. All cases we found consisted of functions with optional basic blocks that can execute before the main function body, and finish by jumping over the main function body prologue. Figure 4 lists such jumps as *multi-entry jumps*, and shows

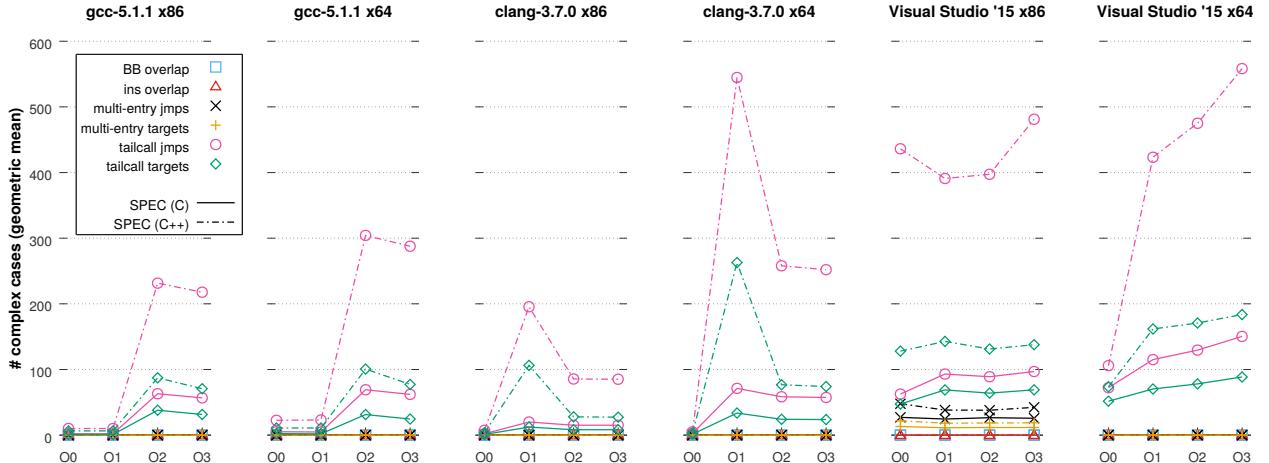


Figure 4: Prevalence of complex constructs in SPEC CPU2006 binaries.

the targeted main function bodies as *multi-entry targets*. In binaries compiled with `gcc` and `clang`, we found up to 18 multi-entry jumps for C code, and up to 64 for C++, with the highest prevalence in x64 binaries. Visual Studio produced up to 172 multi-entry jumps for C, and up to 88 for C++, the construct being most prevalent in x86 code. This kind of multi-entry function is handled well by disassemblers in practice, producing no notable decrease in disassembly accuracy compared to other functions.

Tailcalls form the most prevalent complex case, and do negatively affect function start detection if the target function is never called normally (see Section 3.1.1). The largest number of tailcalls (listed as *tailcall jumps* in Figure 4) is found in `clang` x64 C++ binaries, at a mean of 545 cases. Visual Studio produces a similar number of tailcalls. For `clang`, the number of tailcalls peaks at optimization level 01, while Visual Studio peaks at 03. For `clang` (and to a lesser extent `gcc`), higher optimization levels can lead to a decrease in tailcalls through other modifications like code merging and code elimination.

Jump tables (due to switches) are by far the most common case of inline data. They occur as inline data only on Visual Studio (`gcc` and `clang` place jump tables in the `.rodata` section). As seen in Section 3.1.1, inline data causes false positive instructions especially in linear disassembly (peaking at 0.56% false positives).

Another challenge due to jump tables is locating all case blocks belonging to the switch; these are typically reached indirectly via a jump that loads its target address from the jump table. Linear disassembly covers 100% of case blocks correctly on `gcc` and `clang` (see Section 3.1.1), and also achieves very high accuracy for Visual Studio. The best performing recursive disassemblers, most notably IDA Pro, also achieved very high coverage of switch/case blocks; coverage of these blocks is comparable to the overall instruction/basic block recov-

ery rates. This is because many recursive disassemblers have special heuristics for identifying and parsing standard jump tables.

3.1.4 Optimizing for Size

At optimization levels 00–03, no overlapping or shared basic blocks occur. A reasonable hypothesis is that compilers might more readily produce such blocks when optimizing for size (optimization level 0s) rather than for performance. To verify this, we recompiled the SPEC C and C++ benchmarks with size optimization, and repeated our disassembly tests.

Even for size-optimized binaries, we did not find any overlapping or shared blocks. Moreover, the accuracy of the instruction boundaries, callgraph and ICFG did not significantly differ from our results for 00–03. Function starts and argument lists were comparable in precision to those for performance-optimized binaries (02–03).

3.2 Shared Library Objects

This section discusses our disassembly results and complex case analysis for library code. Libraries are often highly optimized, and therefore contain more complex (handcrafted) corner cases than application code. We focus our analysis on `glibc-2.22`, the standard C library used in GNU systems, compiled in its default configuration (`gcc` with optimization level 02). This is one of the most widespread and highly optimized libraries, and is often cited as a highly complex case [5, 23].

3.2.1 Disassembly Results

Table 2 shows disassembly results for `glibc-2.22`, for all tested disassemblers that support 64-bit ELF binaries. Nearly all disassemblers display significantly lower

	Instructions	Functions	Signatures	ICFG	Callgraph
gcc-5.1.1 x64					
angr	64.4	75.6	—	70.2	87.9
BAP	65.3	79.6	—	72.4	84.8
ByteWeight	—	29.3	—	—	—
Dyninst	79.7	85.2	—	87.6	95.5
Hopper	84.3	93.3	—	90.6	93.9
IDA Pro	96.0	92.0	5.4	99.9	99.9
Linear	99.9	—	—	—	—

Table 2: Disassembly results for glibc (% correct).

accuracy on instruction boundaries than the mean for application binaries in equivalent compiler configurations. Only IDA Pro and linear disassembly are on par with their performance on application code, achieving very good accuracy without any false positives. Note that objdump achieves 99.9% accuracy instead of the usual 100% for ELF binaries. This is because unlike IDA Pro, it does not explicitly separate the overlapping instructions that occur in glibc (see Section 3.2.2).

Function start results are on par with, or even exceed the mean for application binaries; this holds true for all disassemblers. Moreover, the accuracy of function argument lists (5.4%) is much higher than one would expect from the x64 SPEC CPU2006 results (under 1% accuracy). This is because IDA Pro comes with a set of code signatures designed to recognize standard library functions that are statically linked into binaries.

For the ICFG, we see the same pattern as for instructions: all disassemblers perform worse than for application code, while IDA Pro delivers comparable accuracy. Callgraph accuracy is below the mean for most disassemblers, though IDA Pro and Dyninst perform very close to the mean, and BAP well exceeds it.

3.2.2 Complex Constructs

Overall, we found the glibc-2.22 code to be surprisingly well-behaved. Our analysis found no overlapping or shared basic blocks, and no inline data. Indeed, the glibc developers have taken special care to prevent this, explicitly placing data and jump tables in the .rodata section even when manually declared in handwritten assembly code. Prior work has analysed earlier versions of glibc, showing that inline jump tables *are* present in glibc-2.12 [23]. Moreover, inline zero-bytes used for function padding are confirmed in versions up to 2.21. This is worth noting, as older glibc versions may still be encountered in practice. Our analysis of glibc versions ranging from 2.12 to 2.22 shows consistently improving disassembler-friendliness over time.

We did find some complex constructs that do not occur in application code, the most notable being overlapping

```
7b05a: cmpl    $0x0,%fs:0x18
7b063: je     7b066
7b065: lock cmpxchg %rcx,0x3230fa(%rip)
```

Listing 6: Overlapping instruction in glibc-2.22.

```
e9a30 <splice>:
e9a30: cmpl    $0x0,0x2b9da9(%rip)
e9a37: jne     e9a4c <_splice_nocancel+0x13>
e9a39 <_splice_nocancel>:
e9a39: mov     %rcx,%r10
e9a3c: mov     $0x113,%eax
e9a41: syscall
e9a43: cmp     $0xfffffffffffff001,%rax
e9a49: jae     e9a7f <_splice_nocancel+0x46>
e9a4b: retq
e9a4c: sub     $0x8,%rsp
e9a50: callq   f56d0 <__libc_enable_asynccancel>
[...]
```

Listing 7: Multi-entry function in glibc-2.22.

instructions. We found 31 such instructions in glibc. All of these are instructions with optional prefixes, such as the one shown in Listing 6. These overlapping instructions are defined manually in handcrafted assembly code, and typically use a conditional jump to optionally skip a lock prefix. They correspond to frequently cited complex cases in the literature [5, 23].

In addition, we found 508 tailcalls resulting from the compiler’s normal optimization; a number comparable to application binaries of similar size as glibc. We also found significantly more multi-entry functions than in the SPEC benchmarks. Most of these belong to the _nocancel family, explicitly defined in glibc, an example of which is shown in Listing 7. These functions provide optional basic blocks which can be prefixed to the main function body to choose a threadsafe variant of the function. These prefix blocks end by jumping over the prologue of the main function body, a pattern also sometimes seen in application code.

Given that all non-standard complex constructs in glibc are due to handwritten assembly, we manually analyzed all assembly code in libc++ and libstdc++. However, the amount of assembly in these libraries is very limited and revealed no new complex constructs. This suggests that the optimization constructs in glibc are typical for low-level libraries, and less common in higher-level ones such as the C++ standard libraries.

3.3 Static Linking & Linker Optimization

Static linking can reduce disassembler performance on application binaries by merging complex library code into the binary. Link-time optimization performs intermodular optimization at link-time, as opposed to more local compile-time optimizations. It is a relatively new feature that is gaining in popularity, and could worsen disassembler performance if combined with static linking, by optimizing application and library code as a whole. To study

	Instructions	Functions	Signatures	ICFG	Callgraph
gcc-5.1.1 x64 with -static					
SPEC/C 00	96.2	69.4	0.1	98.3	98.2
SPEC/C 01	96.2	68.4	0.2	98.6	98.4
SPEC/C 02	95.5	67.1	0.2	98.8	98.9
SPEC/C 03	95.6	65.7	0.2	98.7	98.7
SPEC/C 0s	95.9	67.8	0.2	98.7	98.4
gcc-5.1.1 x64 with -static and -flio					
SPEC/C 00	96.3	69.3	0.2	98.5	98.3
SPEC/C 01	96.0	68.6	0.3	98.6	98.4
SPEC/C 02	95.0	67.4	0.3	98.3	98.0
SPEC/C 03	95.2	66.9	0.3	98.3	98.4
SPEC/C 0s	95.5	67.8	0.2	98.4	97.7

Table 3: IDA Pro 6.7 disassembly results for static and link-time optimized SPEC C benchmarks (% correct, geometric mean).

the effects of these options, we recompiled the SPEC CPU2006 C benchmarks, statically linking them with `glibc-2.22` using `gcc`'s `-static` flag. Subsequently, we repeated the process with both static linking and link-time optimization (`gcc`'s `-flio`) enabled.

As expected, static linking merges complex cases from `glibc` into SPEC, including overlapping instructions. The effect on disassembly performance is shown in Table 3 for IDA, the overall best performing disassembler in our `glibc` tests. The impact is slight but noticeable, with an instruction accuracy drop of up to 3 percentage points compared to baseline SPEC; about the same as for `glibc`. As can be seen in Table 3, link-time optimization does not significantly decrease disassembly accuracy compared to static linking only.

Function start detection suffers from static linking mostly at lower optimization levels, dropping from a mean of 80% to just under 70% for 00; at level 03 the performance is not significantly reduced. Again, link-time optimization does not worsen the situation compared to pure static linking. For the ICFG and callgraph tests, a small accuracy drop is again seen at lower optimization levels, again with no more adverse effects due to link-time optimization. For instance, ICFG accuracy drops from close to 100% mean in baseline SPEC to just over 98% in statically linked SPEC at 00, while the results at 02 and 03 show no negative impact. We suspect that this is a result of optimized library code being linked in even at lower optimization levels. Overall, we do not expect any significant adverse impact on binary-based research as link-time optimization gains in popularity.

4 Implications of Results

This section discusses the implications of our results for three popular directions in binary-based research: (1)

Control-Flow Integrity, (2) Decompilation, and (3) Automatic bug search. A detailed comparison of our results to assumptions in the literature is given in Section 5.

4.1 Control-Flow Integrity

Control-Flow Integrity (CFI) is currently one of the most popular research directions in systems security, as shown in Table 6. Binary-level CFI typically relies on binary instrumentation to insert control flow protections into proprietary or legacy binaries [1, 10, 24, 29, 41, 45, 46, 48]. Though a wide variety of CFI solutions has been proposed, most of these have similar binary analysis requirements, due to their common aim of protecting indirect jumps, indirect calls, and return instructions. We structure our discussion around what is needed to analyze and protect each of these control edge types.

Indirect calls. Typically, protecting an indirect call requires instrumenting both the call site (the `call` instruction itself, possibly including parameters), and the call target (the called function). Finding call sites relies mainly on accurate and complete disassembly of the basic instructions. As shown in Figure 2a, these can be recovered with extremely high accuracy, even 100% accuracy for linear disassembly on `gcc` and `clang` binaries. Thus, a binary-level CFI solution is unlikely to encounter problems analyzing and instrumenting call sites.

For Visual Studio binaries, there is a chance that a small percentage of call sites may be missed. Depending on the specific CFI solution, it may be possible to detect calls from uninstrumented sites in the target function, triggering a runtime error handling mechanism (see Section 5). Since these cases are rare, it is then feasible to perform more elaborate (slow path) alternative security checks.

The main challenge is to accurately detect all possible target functions for each indirect call. As a basic prerequisite, this requires finding the complete set of indirectly called functions. As shown in Section 3.1.1 and Figure 2b, this is one of the most challenging problems in disassembly — at high optimization levels, 20% or more of all functions are routinely missed.

Moreover, fine-grained CFI systems must perform even more elaborate analysis to decide which functions are legal targets for each indirect call site. Overestimating the set of legal targets leads to attacks which redirect indirect calls to unexpected functions [12]. Matching call sites to a set of targets typically requires an accurate (I)CFG, so that control-flow and data-flow analysis can be performed to determine which function pointers are passed to each call site. Figure 2d and Sections 3.1.1–3.1.3 show that an accurate and complete ICFG is typically available, including accurate resolution of switch/jump tables in the best disassemblers. Although this type of analysis remains

extremely challenging, especially if done interprocedurally (requiring accurate indirect call resolution), it is at least not limited by the accuracy of basic blocks or direct control edges.

Additionally, fine-grained CFI systems can benefit from function signature information, to further narrow down the set of targets per call site by matching the function prototype to parameters passed at the call site [39]. Though signature information is often far from complete (Figure 2c), especially on x64, the information which is available can still be useful — even with incomplete information, the target set can be reduced, directly leading to security improvements. However, care must be taken to make the analysis as conservative as possible; if this is not done, the inaccuracy of function signature information can easily cause illegal function calls to be allowed, or worse, can cause legal calls to be inadvertently blocked.

Indirect jumps. Protecting indirect jumps requires analysis similar to the requirements for indirect calls. However, as indirect jumps are typically intraprocedural, protecting them usually does not rely on function detection. Instead, accurate switch/jump table resolution is required, which is available in disassemblers like IDA Pro (Section 3.1.3).

Return instructions. Return instructions are typically protected using a shadow stack, which requires instrumenting all call and return sites (and jumps, to handle tailcalls) [8]. Given the accurate instruction recovery possible with modern disassemblers (Figure 2a), it is possible to accurately and completely instrument these sites.

Summarizing, the main challenge for modern CFI lies in accurately and completely protecting indirect call sites. The reasons for this are twofold: (1) Function detection is one of the most inaccurate primitives (especially for indirectly called functions), even in state of the art disassemblers, and (2) It is currently very difficult to recover rich information, such as function signature information, through disassembly. This makes it extremely challenging to accurately couple indirect call sites with valid targets.

4.2 Decompilation

Instead of translating a binary into assembly instructions, decompilers lift binaries to a higher-level language, typically (pseudo-) C. Decompilers are typically built on top of a disassembler, and therefore rely heavily on the quality of the disassembly [33, 44].

As most decompilers operate at function granularity, they rely on accurate function start information. Moreover, they must translate all basic blocks belonging to a function, requiring knowledge of the function’s CFG. In effect, this requires not only accurate function start

detection, but accurate function boundary detection. As described in related work, function boundary detection is even more challenging than function start detection, as it additionally requires locating the end address of each function [4]. This is difficult, especially in optimized binaries, where tailcalls often blur the boundaries between functions (since the `jmp` instructions used in tailcalls can easily be mistaken for intraprocedural control transfers).

In addition to function detection, decompilers rely on accurate instruction disassembly, and can also greatly benefit from function signature/type information. Moreover, switch detection is required to correctly attribute all switch case blocks to their parent function. Finally, call-graph information is useful to understand the connections between decompiled functions.

The impact of inaccuracies for decompilation is not as severe as for CFI systems, since decompiled code is typically intended for use in manual reverse engineering rather than automated analysis. However, disassembly errors can still affect the decompilation process itself, especially in later passes (such as stack frame analysis or data type analysis passes) over the raw decompiled function. Such analysis phases, as well as human reverse engineers, must take into account the high probability of errors in function boundary and signature information.

4.3 Automatic Bug Search

The binary analysis requirements of automatic bug search systems depend on the type of bug being searched for, and the granularity of the search. In practice, many such systems operate at the function level, both for ease of analysis, and because it is a suitable search-granularity for common bugs, such as stack-based bugs [14, 27, 50]. Operating at the function level is also useful for interoperability with other binary analysis primitives, such as symbolic execution, which are powerful tools for semantic analysis but do not scale to full binaries [14].

Thus, like decompilation, many automatic bug search systems rely on accurate function boundary information and per-function CFGs. Fortunately, despite the relatively large inaccuracies in the input information, the output of bug detection systems tends to degrade gracefully — input inaccuracies may lead to bugs being missed, but typically do not affect the correctness of the analysis for other parts of the code. Quantifying the accuracy of the inputs (disassembly, CFG, function boundaries) helps users to determine the expected output completeness of automatic bug search systems.

5 Disassembly in the Literature

Given our disassembly results, we studied recent binary-based research to determine how well the capabilities

	# Papers	Instructions	Functions	Signatures	CFG	Callgraph
angr	0	0	0	0	0	0
BAP	2	1	2	1	2	0
ByteWeight	0	0	0	0	0	0
Dyninst	1	1	0	0	1	1
Hopper	0	0	0	0	0	0
IDA Pro	13	11	6	2	11	4
Jakstab	0	0	0	0	0	0
PSI/BinCFI	4	3	3	0	3	2
Linear	2	2	1	0	1	1
Other/Custom	8	7	2	0	6	3
Total	30	25	14	3	24	11

Table 4: Primitives/disassemblers used in the literature.

of disassemblers match the expectations in the literature. Our study covers research published between 2013 and 2015 in all top-tier systems security conferences, namely S&P (Oakland), CCS, NDSS and USENIX Security. In addition, we cover research published in these same years at RAID and ACSAC, two other major conferences which are popular targets for such research.

We found 30 papers on binary-based research published in these venues, summarized in Table 6. The rest of this section presents aggregated findings to provide a degree of anonymization for these papers.

Table 4 shows the primitives and disassemblers used in these papers. IDA Pro is by far the most popular, for all primitives; our disassembly results (Section 3) justify this choice. Despite its good accuracy, linear disassembly is among the least used, even for papers that handle only ELF binaries. This may result from the widespread belief that inline data causes far more problems than we found.

Instructions are the most often needed primitive, used by 25 of the 30 papers. It is followed by the CFG (24 papers) and function starts (14 papers). Function signature information is needed by only 3 of the analyzed papers. One paper used linear disassembly as a basis for building a CFG and callgraph, and scanning for function starts.

Table 5 provides a more detailed insight into the properties of the papers we analyzed. We distinguish between papers that target Windows PE binaries, and those that target Linux ELF. This is because some complex cases, such as inline data, are more often generated by Visual Studio, deserving closer attention in Windows papers.

Most papers that support obfuscated binaries target Windows (33% of papers versus 10% for Linux). This is because obfuscation typically occurs in malware, which is more prevalent on Windows. Though we do not consider obfuscated binaries in our tests, it is still interesting to know how many papers target such binaries. After all, these papers should pay special attention to disassembly errors and complex corner cases. Unfortunately, this is not the case; only 50% of papers that support obfuscation discuss potential errors, while 33% implement error

Property	Subproperty	All papers		Top-tier	
		#	%	#	%
Windows PE x86/x64 (16 papers, 12 top-tier)					
Obfuscated code		5	31%	4	33%
Optimized binaries		14	88%	11	92%
Stripped binaries		15	94%	11	92%
Recursive disassembly		16	100%	12	100%
Needs relocation info		2	12%	2	17%
Primitive errors discussed	Instructions	5 (13)	38%	5 (9)	56%
	Functions	1 (5)	20%	1 (4)	25%
	Signatures	0 (2)	0%	0 (2)	0%
	Callgraph	4 (5)	80%	4 (5)	80%
	CFG	5 (13)	38%	5 (10)	50%
Complex cases discussed		5	31%	5	42%
Primitive errors handled	Overestimate	4	25%	4	33%
	Underestimate	3	19%	2	17%
	Runtime	1	6%	1	8%
Errors are fatal		13	81%	11	92%
Linux ELF x86/x64 (14 papers, 10 top-tier)					
Obfuscated code		1	7%	1	10%
Optimized binaries		13	93%	9	90%
Stripped binaries		11	79%	7	70%
Recursive disassembly		12	86%	8	80%
Primitive errors discussed	Instructions	6 (12)	50%	6 (9)	67%
	Functions	3 (9)	33%	3 (6)	50%
	Signatures	1 (1)	100%	1 (1)	100%
	Callgraph	2 (6)	33%	2 (4)	50%
	CFG	5 (11)	45%	5 (8)	62%
Complex cases discussed		1	7%	1	10%
Primitive errors handled	Overestimate	4	29%	3	30%
	Underestimate	0	0%	0	0%
	Runtime	1	7%	1	10%
Errors are fatal		8	57%	6	60%

Table 5: Properties of binary-based papers (number and percentage of papers). Numbers in parentheses indicate the total number of papers that use this primitive.

handling. This is no better than the overall number. Moreover, only 17% of these papers explicitly discuss complex cases; far below the overall rate for Windows.

Nearly all papers support optimized binaries (90% or more for both Linux and Windows, overall as well as top-tier). Stripped binaries are supported by an equally large majority of papers on Windows, and by a slightly smaller majority on Linux. Curiously, the number of top-tier papers that support stripped binaries on Linux (70%) is significantly less than the overall number (79%).

The vast majority of papers use recursive disassembly (100% on Windows and 86% on Linux), with IDA Pro being the most popular disassembler. The few papers that do use linear disassembly are based on objdump, and augment it with a layer of error correction. Interestingly, these papers claim perfect (100% accurate) or close to perfect disassembly. As shown in Section 3.1.1, this precision on Linux binaries owes entirely to the core linear disassembly, making any error correction redundant other than for a few corner cases in library code (and obfuscated code, which these papers do not consider).

A relatively small percentage of Windows papers use relocation information to find disassembly starting points. At 17%, this number is slightly higher for top-tier papers.

Discussion on disassembly errors and complex cases is somewhat lacking in the analyzed papers. For most prim-

Title	Authors	Venue	Year	Top-tier
<i>A Principled Approach for ROP Defense</i> [30]	Qiao et al.	AC SAC	2015	
<i>Binary Code Continent: Finer-Grained Control Flow Integrity (...)</i> [41]	Wang et al.	AC SAC	2015	
<i>Blanket Execution: Dynamic Similarity Testing for Program (...)</i> [11]	Egele et al.	USENIX Sec	2014	✓
<i>BYTEWEIGHT: Learning to Recognize Functions in Binary Code</i> [4]	Bao et al.	USENIX Sec	2014	✓
<i>CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying (...)</i> [6]	Bonfante et al.	CCS	2015	✓
<i>Control Flow and Code Integrity for COTS binaries</i> [49]	Zhang et al.	AC SAC	2015	
<i>Control Flow Integrity for COTS Binaries</i> [48]	Zhang et al.	USENIX Sec	2013	✓
<i>Cross-Architecture Bug Search in Binary Executables</i> [27]	Pewny et al.	S&P	2015	✓
<i>DUET: Integration of Dynamic and Static Analyses for Malware (...)</i> [15]	Hu et al.	AC SAC	2013	
<i>Dynamic Hooks: Hiding Control Flow Changes within (...)</i> [40]	Vogl et al.	USENIX Sec	2014	✓
<i>Hardware-Assisted Fine-Grained Control-Flow Integrity (...)</i> [10]	Davi et al.	RAID	2015	
<i>Heisenbyte: Thwarting Memory Disclosure Attacks using (...)</i> [37]	Tang et al.	CCS	2015	✓
<i>High Accuracy Attack Provenance via Binary-based (...)</i> [20]	Hyung Lee et al.	NDSS	2013	✓
<i>Improving Accuracy of Static Integer Overflow Detection in Binary</i> [50]	Zhang et al.	RAID	2015	
<i>Leveraging Semantic Signatures for Bug Search in Binary Programs</i> [28]	Pewny et al.	AC SAC	2014	
<i>Native x86 Decomposition Using Semantics-Preserving (...)</i> [33]	Schwartz et al.	USENIX Sec	2013	✓
<i>No More Gotos: Decompilation Using Pattern-Independent (...)</i> [44]	Yakdan et al.	NDSS	2015	✓
<i>Opaque Control-Flow Integrity</i> [24]	Mohan et al.	NDSS	2015	✓
<i>Oxymoron Making Fine-Grained Memory Randomization Practical (...)</i> [2]	Backes et al.	USENIX Sec	2014	✓
<i>Practical Context-Sensitive CFI</i> [1]	Andriesse et al.	CCS	2015	✓
<i>Practical Control Flow Integrity & Randomization for (...)</i> [46]	Zhang et al.	S&P	2013	✓
<i>Reassemblable Disassembling</i> [42]	Wang et al.	USENIX Sec	2015	✓
<i>Recognizing Functions in Binaries with Neural Networks</i> [35]	Chul et al.	USENIX Sec	2015	✓
<i>ROPecker: A Generic and Practical Approach for Defending (...)</i> [9]	Cheng et al.	NDSS	2014	✓
<i>StackArmor: Comprehensive Protection from Stack-based (...)</i> [8]	Chen et al.	NDSS	2015	✓
<i>Towards Automated Integrity Protection of C++ Virtual Function (...)</i> [13]	Gawlik et al.	AC SAC	2014	
<i>Towards Automatic Software Lineage Inference</i> [16]	Jang et al.	USENIX Sec	2013	✓
<i>vGuard: Strict Protection for Virtual Function Calls (...)</i> [29]	Prakash et al.	NDSS	2015	✓
<i>VTint: Protecting Virtual Function Tables' Integrity</i> [45]	Zhang et al.	NDSS	2015	✓
<i>X-Force: Force-Executing Binary Programs for Security (...)</i> [26]	Peng et al.	USENIX Sec	2014	✓

Table 6: Set of papers discussed in the literature study.

itives on Windows, at best 50% of papers discuss what happens if the primitive is not recovered perfectly. This number applies to the top-tier papers; overall, the number is even lower. The number for Linux-based papers is slightly better, though even here only a small majority of papers devote significant attention to potential problems. One would expect more thorough discussion, especially given that between 80% and 90% of Windows papers, and around 60% of Linux papers, may suffer malignant failures given imperfect primitives. The issue is most apparent in the Windows papers that require function start information. Only 25% of the top-tier papers that require function starts consider potential errors in this information, even though Section 3.1.1 shows that function starts are quite challenging to recover accurately.

The percentage of Windows papers that discuss complex cases such as inline data varies from 31% overall to 42% for top-tier papers. Again, this is less than we would expect given the prevalence of inline jump tables generated by Visual Studio. The number for papers that target Linux is even lower, though this causes fewer issues as complex cases in ELF binaries are rare.

There is a strong correlation within all papers between discussion of errors and complex cases, and support for error handling. Papers that discuss such cases also tend to implement some mechanism for dealing with errors if they occur. Conversely, papers that do not implement error handling nearly always fail to discuss errors at all.

We identified three popular and recurring categories of error handling mechanisms.

(1) *Overestimation*: For instance, CFG and callgraph overestimation are popular in papers that build binary-level security; it minimizes the risk of accidentally prohibiting valid edges, though the precision of security policies may suffer slightly.

(2) *Underestimation*: This is used in papers where soundness is more important than completeness.

(3) *Runtime augmentation*: Some papers use static analysis to approximate a primitive, and use low-cost runtime checks to fix errors in the primitive where needed.

Overestimation is the most popular error handling strategy, used in around 30% of top-tier papers. It is followed by underestimation and runtime augmentation.

6 Discussion

Our findings show a dualism in the stance on disassembly in the literature. On the one hand, the difficulty of pure (instruction-level) disassembly is often exaggerated. The prevalence of complex constructs like overlapping basic blocks, inline data, and overlapping instructions is frequently overestimated, especially for `gcc` and `clang` [5, 23]. This leads reviewers and researchers to underestimate the effectiveness of binary-based research.

We showed that unless binaries are deliberately obfuscated, instruction recovery is extremely accurate, es-

pecially in ELF binaries generated with `gcc` or `clang`. We did not find any inline data for these binaries, even in optimized library code; even jump tables are explicitly placed in the `.rodata` section. Moreover, in Visual Studio binaries with jump tables in the code section, modern disassemblers like IDA Pro recognize and resolve them quite accurately. The rare overlapping instructions in handcrafted library code take on a limited number of forms, typically using a direct conditional jump over a prefix. These are resolved without problems by IDA Pro and Dyninst, among others. The same is true for multi-entry functions, which are also rare. Moreover, overlapping/shared basic blocks (commonly cited as particularly challenging for binary analysis), do not appear in our findings at all.

On the other hand, some primitives really do often suffer from inaccuracies. Some recursive disassemblers used for binary instrumentation (notably Dyninst) regularly miss up to 10% of basic blocks in optimized binaries, calling for special attention in systems which rely on basic block-level binary instrumentation. Additionally, function signatures in 64-bit code are extremely inaccurate; fortunately, they are also rarely used in the literature.

However, function starts *are* regularly needed, though the false negative rate regularly rises to 20% or more even for the best performing disassemblers. This is especially true in optimized binaries, or in coding styles that make extensive use of function pointers. Worse, false positive function starts are almost as common. This can lead to problems in some binary-based research, especially binary instrumentation, if care is not taken to ensure graceful failure in the event of misdetected function starts. Symbols offer a great deal of help, especially in reducing the false negative rate. Unfortunately, they are rarely available in practice.

It is surprising then, to find that only 20% to 25% (top-tier) of Windows papers that use function starts, and 33% to 50% (top-tier) of the Linux papers, devote any attention to discussing these problems. A similarly small number of papers implement error handling, even though errors can cause malignant failures in a majority of papers. While it is not impossible to base well-functioning binary-based systems on function start information (or other primitives), it is crucial that such work implement mechanisms for handling inaccuracies. Three effective classes of error handling (depending on the situation) have already been proposed in the literature: overestimation, underestimation, and runtime augmentation.

We hope our study will facilitate a better match between expectations on disassembly in future research, and the performance actually delivered by modern disassemblers. Moreover, we believe our findings can be used to better judge where problems are to be expected, and to implement effective mechanisms for dealing with them.

7 Related Work

Prior work on disassembly precision focused on complex corner cases [5, 23, 25] or obfuscated code [18, 34], showing that these can strongly reduce disassembly accuracy. We focus instead on the performance of modern disassemblers given realistic full-scale binaries without active anti-disassembly techniques.

Miller et al. center their analysis around complex cases in `glibc-2.12` [23]. Their findings largely correspond to our own, though we found no inline jump tables in `glibc-2.22`. In addition to their `glibc` analysis, Miller et al. find complex cases in SPEC CPU2006; however, this analysis focuses exclusively on statically linked binaries. We show in Section 3.3 that these cases are entirely due to embedded library code, and are extremely rare in non-statically linked applications.

Our finding that function starts are among the most challenging primitives to recover is in agreement with results by Bao et al. [4].

Paleari et al. study instruction decoders in disassemblers [25], which parse individual x86 instructions. Specific instructions that are sometimes wrongly parsed have also been outlined by the authors of Capstone [31].

Complex constructs in obfuscated code are discussed by Schwarz et al. [34], Linn et al. [21] and Kruegel et al. [18]. We show that these worst-case complex constructs are exceedingly rare in non-obfuscated code.

8 Conclusion

Our study contradicts the widespread belief that complex constructs severely limit the usefulness of binary-based research. Instead, we show that modern disassemblers achieve close to 100% instruction disassembly accuracy for compiler-generated binaries, and that constructs like inline data and overlapping code are very rare. Errors in areas where disassembly is truly lacking, such as function start recovery, are not discussed nearly as often in the literature. By analyzing discrepancies between disassembler capabilities and the literature, our work provides a foundation for guiding future research.

Acknowledgements

We thank the anonymous reviewers for their valuable input to improve the paper. We also thank Mingwei Zhang and Rui Qiao for their proofreading and feedback. This work was supported by the European Commission through project H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 644571, and by the Netherlands Organisation for Scientific Research through grant NWO CSI-DHS 628.001.021 and the NWO 639.023.309 VICI “Dowsing” project.

References

- [1] ANDRIESSE, D., VAN DER VEEN, V., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. Practical Context-Sensitive CFI. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)* (Denver, CO, USA, October 2015), ACM.
- [2] BACKES, M., AND NÜRNBERGER, S. Oxymoron Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).
- [3] BALAKRISHNAN, G., AND REPS, T. WYSIWYX: What You See is Not What You eXecute. *ACM Transactions on Programming Languages and Systems* 32, 6 (Aug. 2010), 23:1–23:84.
- [4] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).
- [5] BERNAT, A. R., AND MILLER, B. P. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools* (2011).
- [6] BONFANTE, G., FERNANDEZ, J., MARION, J.-Y., ROUXEL, B., SABATIER, F., AND THIERRY, A. CoDisasm: Medium Scale Conicatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)* (2015).
- [7] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)* (2011).
- [8] CHEN, X., SLOWINSKA, A., ANDRIESSE, D., BOS, H., AND GIUFFRIDA, C. StackArmor: Comprehensive Protection from Stack-Based Memory Error Vulnerabilities for Binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (San Diego, CA, USA, February 2015), Internet Society.
- [9] CHENG, Y., ZHOU, Z., YU, M., DING, X., AND DENG, R. H. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)* (2014).
- [10] DAVI, L., KOEBERL, P., AND SADEGHI, A.-R. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)* (2015).
- [11] EGELE, M., WOO, M., CHAPMAN, P., AND BRUMLEY, D. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).
- [12] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUKOS, S. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)* (Denver, CO, USA, 2015), ACM.
- [13] GAWLIK, R., AND HOLZ, T. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)* (2014).
- [14] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)* (2013).
- [15] HU, X., AND SHIN, K. G. DUET: Integration of Dynamic and Static Analyses for Malware Clustering with Cluster Ensembles. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)* (2013).
- [16] JANG, J., WOO, M., AND BRUMLEY, D. Towards Automatic Software Lineage Inference. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)* (2013).
- [17] KINDER, J. *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, 2010.
- [18] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium (USENIX Sec'04)* (2004).
- [19] LAURENZANO, M., TIKIR, M. M., CARRINGTON, L., AND SNAVELY, A. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (2010).
- [20] LEE, K. H., ZHANG, X., AND XU, D. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'13)* (2013).
- [21] LINN, C., AND DEBRAY, S. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)* (2003).
- [22] MICROSOFT DEVELOPER NETWORK. Overview of x64 Calling Conventions, 2015. <https://msdn.microsoft.com/en-us/library/ms235286.aspx>.
- [23] MILLER, B. P., AND MENG, X. Binary Code is Not Easy, 2015. Technical report, University of Wisconsin-Madison.
- [24] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., AND FRANZ, M. Opaque Control-Flow Integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (2015).
- [25] PALEARI, R., MARTIGNONI, L., FRESI ROGLIA, G., AND BRUSCHI, D. N-Version Disassembly: Differential Testing of x86 Disassemblers. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (2010), ISSTA'10.
- [26] PENG, F., DENG, Z., ZHANG, X., XU, D., LIN, Z., AND SU, Z. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).
- [27] PEWNY, J., GARMANY, B., GAWLIK, R., ROSSOW, C., AND HOLZ, T. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)* (2015).
- [28] PEWNY, J., SCHUSTER, F., ROSSOW, C., BERNHARD, L., AND HOLZ, T. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)* (2014).

- [29] PRAKASH, A., HU, X., AND YIN, H. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (San Diego, CA, USA, February 2015), Internet Society.
- [30] QIAO, R., ZHANG, M., AND SEKAR, R. A Principled Approach for ROP Defense. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)* (2015).
- [31] QUYNH, N. A. Capstone: Next-Gen Disassembly Framework. In *Blackhat USA* (2014).
- [32] ROMER, T., VOELKER, G., LEE, D., WOLMAN, A., WONG, W., LEVY, H., BERSHAD, B., AND CHEN, B. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop (NT'97)* (1997).
- [33] SCHWARTZ, E. J., LEE, J., WOO, M., AND BRUMLEY, D. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)* (2013).
- [34] SCHWARZ, B., DEBRAY, S., AND ANDREWS, G. Disassembly of Executable Code Revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)* (2002).
- [35] SHIN, E. C. R., SONG, D., AND MOAZZEZI, R. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium (USENIX Sec'15)* (2015).
- [36] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.
- [37] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)* (2015).
- [38] TRAIL OF BITS. A Preview of McSema, 2014. Technical report. <http://blog.trailofbits.com/2014/06/23/a-preview-of-mcsema/>.
- [39] VAN DER VEEN, V., GÖKTAŞ, E., CONTAG, M., PAWLOSKI, A., CHEN, X., RAWAT, S., BOS, H., HOLZ, T., ATHANASOPOULOS, E., AND GIUFFRIDA, C. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proceedings of the 37th Symposium on Security and Privacy (S&P'16)* (May 2016).
- [40] VOGL, S., GAWLIK, R., GARMANY, B., KITTEL, T., PFOH, J., ECKERT, C., AND HOLZ, T. Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).
- [41] WANG, M., YIN, H., BHASKAR, A. V., SU, P., AND FENG, D. Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)* (2015).
- [42] WANG, S., WANG, P., AND WU, D. Reassemblable Disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Sec'15)* (2015).
- [43] WARTELL, R., ZHOU, Y., HAMLEN, K. W., KANTARIOGLU, M., AND THURAISINGHAM, B. M. Differentiating Code from Data in x86 Binaries. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECDL'11)* (2011).
- [44] YAKDAN, K., ESCHWEILER, S., GERHARDS-PADILLA, E., AND SMITH, M. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (2015).
- [45] ZHANG, C., SONG, C., CHEN, K. Z., CHEN, Z., AND SONG, D. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (2015).
- [46] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)* (2013).
- [47] ZHANG, M., QIAO, R., HASABNI, N., AND SEKAR, R. A Platform for Secure Static Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14)* (2014).
- [48] ZHANG, M., AND SEKAR, R. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)* (2013).
- [49] ZHANG, M., AND SEKAR, R. Control Flow and Code Integrity for COTS binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)* (2015).
- [50] ZHANG, Y., SUN, X., DENG, Y., CHENG, L., ZENG, S., FU, Y., AND FENG, D. Improving Accuracy of Static Integer Overflow Detection in Binary. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)* (2015).

Stealing Machine Learning Models via Prediction APIs

Florian Tramèr
EPFL

Fan Zhang
Cornell University

Ari Juels
Cornell Tech, Jacobs Institute

Michael K. Reiter
UNC Chapel Hill

Thomas Ristenpart
Cornell Tech

Abstract

Machine learning (ML) models may be deemed confidential due to their sensitive training data, commercial value, or use in security applications. Increasingly often, confidential ML models are being deployed with publicly accessible query interfaces. ML-as-a-service (“predictive analytics”) systems are an example: Some allow users to train models on potentially sensitive data and charge others for access on a pay-per-query basis.

The tension between model confidentiality and public access motivates our investigation of *model extraction attacks*. In such attacks, an adversary with black-box access, but no prior knowledge of an ML model’s parameters or training data, aims to duplicate the functionality of (i.e., “steal”) the model. Unlike in classical learning theory settings, ML-as-a-service offerings may accept partial feature vectors as inputs and include confidence values with predictions. Given these practices, we show simple, efficient attacks that extract target ML models with near-perfect fidelity for popular model classes including logistic regression, neural networks, and decision trees. We demonstrate these attacks against the online services of BigML and Amazon Machine Learning. We further show that the natural countermeasure of omitting confidence values from model outputs still admits potentially harmful model extraction attacks. Our results highlight the need for careful ML model deployment and new model extraction countermeasures.

1 Introduction

Machine learning (ML) aims to provide automated extraction of insights from data by means of a predictive model. A predictive model is a function that maps feature vectors to a categorical or real-valued output. In a supervised setting, a previously gathered data set consisting of possibly confidential feature-vector inputs (e.g., digitized health records) with corresponding output class labels (e.g., a diagnosis) serves to train a predictive model

that can generate labels on future inputs. Popular models include support vector machines (SVMs), logistic regressions, neural networks, and decision trees.

ML algorithms’ success in the lab and in practice has led to an explosion in demand. Open-source frameworks such as PredictionIO and cloud-based services offered by Amazon, Google, Microsoft, BigML, and others have arisen to broaden and simplify ML model deployment.

Cloud-based ML services often allow model owners to charge others for queries to their commercially valuable models. This pay-per-query deployment option exemplifies an increasingly common tension: The query interface of an ML model may be widely accessible, yet the model itself and the data on which it was trained may be proprietary and confidential. Models may also be privacy-sensitive because they leak information about training data [4, 23, 24]. For security applications such as spam or fraud detection [9, 29, 36, 55], an ML model’s confidentiality is critical to its utility: An adversary that can learn the model can also often evade detection [4, 36].

In this paper we explore *model extraction attacks*, which exploit the tension between query access and confidentiality in ML models. We consider an adversary that can query an ML model (a.k.a. a prediction API) to obtain predictions on input feature vectors. The model may be viewed as a black box. The adversary may or may not know the model type (logistic regression, decision tree, etc.) or the distribution over the data used to train the model. The adversary’s goal is to extract an equivalent or near-equivalent ML model, i.e., one that achieves (close to) 100% agreement on an input space of interest.

We demonstrate successful model extraction attacks against a wide variety of ML model types, including decision trees, logistic regressions, SVMs, and deep neural networks, and against production ML-as-a-service (MLaaS) providers, including Amazon and BigML.¹ In nearly all cases, our attacks yield models that are func-

¹We simulated victims by training models in our own accounts. We have disclosed our results to affected services in February 2016.

Service	Model Type	Data set	Queries	Time (s)
Amazon	Logistic Regression	Digits	650	70
	Logistic Regression	Adult	1,485	149
BigML	Decision Tree	German Credit	1,150	631
	Decision Tree	Steak Survey	4,013	2,088

Table 1: **Results of model extraction attacks on ML services.** For each target model, we report the number of prediction queries made to the ML API in an attack that extracts a 100% equivalent model. The attack time is primarily influenced by the service’s prediction latency ($\approx 100\text{ms}/\text{query}$ for Amazon and $\approx 500\text{ms}/\text{query}$ for BigML).

tionally very close to the target. In some cases, our attacks extract the exact parameters of the target (e.g., the coefficients of a linear classifier or the paths of a decision tree). For some targets employing a model type, parameters or features unknown to the attacker, we additionally show a successful preliminary attack step involving reverse-engineering these model characteristics.

Our most successful attacks rely on the information-rich outputs returned by the ML prediction APIs of all cloud-based services we investigated. Those of Google, Amazon, Microsoft, and BigML all return *high-precision confidence values in addition to class labels*. They also respond to partial queries lacking one or more features. Our setting thus differs from traditional learning-theory settings [3, 7, 8, 15, 30, 33, 36, 53] that assume only *membership queries*, outputs consisting of a class label only. For example, for logistic regression, the confidence value is a simple log-linear function $1/(1+e^{-(\mathbf{w} \cdot \mathbf{x} + \beta)})$ of the d -dimensional input vector \mathbf{x} . By querying $d+1$ random d -dimensional inputs, an attacker can with high probability solve for the unknown $d+1$ parameters \mathbf{w} and β defining the model. We emphasize that while this model extraction attack is simple and non-adaptive, it affects all of the ML services we have investigated.

Such equation-solving attacks extend to multiclass logistic regressions and neural networks, but do not work for decision trees, a popular model choice. (BigML, for example, initially offered only decision trees.) For decision trees, a confidence value reflects the number of training data points labeled correctly on an input’s path in the tree; simple equation-solving is thus inapplicable. We show how confidence values can nonetheless be exploited as pseudo-identifiers for paths in the tree, facilitating discovery of the tree’s structure. We demonstrate successful model extraction attacks that use adaptive, iterative search algorithms to discover paths in a tree.

We experimentally evaluate our attacks by training models on an array of public data sets suitable as stand-ins for proprietary ones. We validate the attacks locally using standard ML libraries, and then present case studies on BigML and Amazon. For both services, we show computationally fast attacks that use a small number of queries to extract models matching the targets on 100% of tested inputs. See Table 1 for a quantitative summary.

Having demonstrated the broad applicability of model extraction attacks to existing services, we consider the most obvious potential countermeasure ML services might adopt: Omission of confidence values, i.e., output of class labels only. This approach would place model extraction back in the membership query setting of prior work in learning theory [3, 8, 36, 53]. We demonstrate a generalization of an adaptive algorithm by Lowd and Meek [36] from binary linear classifiers to more complex model types, and also propose an attack inspired by the agnostic learning algorithm of Cohn et al. [18]. Our new attacks extract models matching targets on $>99\%$ of the input space for a variety of model classes, but need up to $100\times$ more queries than equation-solving attacks (specifically for multiclass linear regression and neural networks). While less effective than equation-solving, these attacks remain attractive for certain types of adversary. We thus discuss further ideas for countermeasures.

In summary, we explore model extraction attacks, a practical kind of learning task that, in particular, affects emerging cloud-based ML services being built by Amazon, Google, Microsoft, BigML, and others. We show:

- *Simple equation-solving model extraction attacks* that use non-adaptive, random queries to solve for the parameters of a target model. These attacks affect a wide variety of ML models that output confidence values. We show their success against Amazon’s service (using our own models as stand-ins for victims’), and also report successful reverse-engineering of the (only partially documented) model type employed by Amazon.
- *A new path-finding algorithm for extracting decision trees* that abuses confidence values as quasi-identifiers for paths. To our knowledge, this is the first example of practical “exact” decision tree learning. We demonstrate the attack’s efficacy via experiments on BigML.
- *Model extraction attacks against models that output only class labels*, the obvious countermeasure against extraction attacks that rely on confidence values. We show slower, but still potentially dangerous, attacks in this setting that build on prior work in learning theory.

We additionally make a number of observations about the implications of extraction. For example, attacks against Amazon’s system indirectly leak various summary statistics about a private training set, while extraction against kernel logistic regression models [57] recovers significant information about individual training data points.

The source code for our attacks is available online at <https://github.com/ftramer/Steal-ML>.

2 Background

For our purposes, a ML model is a function $f: \mathcal{X} \rightarrow \mathcal{Y}$. An input is a d -dimensional vector in the feature space

$\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \dots \times \mathcal{X}_d$. Outputs lie in the range \mathcal{Y} .

We distinguish between categorical features, which assume one of a finite set of values (whose set size is the arity of the feature), and continuous features, which assume a value in a bounded subset of the real numbers. Without loss of generality, for a categorical feature of arity k , we let $\mathcal{X}_i = \mathbb{Z}_k$. For a continuous feature taking values between bounds a and b , we let $\mathcal{X}_i = [a, b] \subset \mathbb{R}$.

Inputs to a model may be pre-processed to perform feature extraction. In this case, inputs come from a space \mathcal{M} , and feature extraction involves application of a function $\text{ex}: \mathcal{M} \rightarrow \mathcal{X}$ that maps inputs into a feature space. Model application then proceeds by composition in the natural way, taking the form $f(\text{ex}(M))$. Generally, feature extraction is many-to-one. For example, M may be a piece of English language text and the extracted features counts of individual words (so-called “bag-of-words” feature extraction). Other examples are input scaling and one-hot-encoding of categorical features.

We focus primarily on classification settings in which f predicts a nominal variable ranging over a set of classes. Given c classes, we use as class labels the set \mathbb{Z}_c . If $\mathcal{Y} = \mathbb{Z}_c$, the model returns only the predicted class label. In some applications, however, additional information is often helpful, in the form of real-valued measures of confidence on the labels output by the model; these measures are called *confidence values*. The output space is then $\mathcal{Y} = [0, 1]^c$. For a given $\mathbf{x} \in \mathcal{X}$ and $i \in \mathbb{Z}_c$, we denote by $f_i(\mathbf{x})$ the i^{th} component of $f(\mathbf{x}) \in \mathcal{Y}$. The value $f_i(\mathbf{x})$ is a model-assigned probability that \mathbf{x} has associated class label i . The model’s predicted class is defined by the value $\text{argmax}_i f_i(\mathbf{x})$, i.e., the most probable label.

We associate with \mathcal{Y} a distance measure $d_{\mathcal{Y}}$. We drop the subscript \mathcal{Y} when it is clear from context. For $\mathcal{Y} = \mathbb{Z}_c$ we use 0-1 distance, meaning $d(y, y') = 0$ if $y = y'$ and $d(y, y') = 1$ otherwise. For $\mathcal{Y} = [0, 1]^c$, we use the 0-1 distance when comparing predicted classes; when comparing class probabilities directly, we instead use the total variation distance, given by $d(\mathbf{y}, \mathbf{y}') = \frac{1}{2} \sum |y[i] - y'[i]|$. In the rest of this paper, unless explicitly specified otherwise, $d_{\mathcal{Y}}$ refers to the 0-1 distance over class labels.

Training algorithms. We consider models obtained via supervised learning. These models are generated by a training algorithm \mathcal{T} that takes as input a training set $\{(\mathbf{x}_i, y_i)\}_i$, where $(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ is an input with an associated (presumptively correct) class label. The output of \mathcal{T} is a model f defined by a set of *parameters*, which are model-specific, and *hyper-parameters*, which specify the type of models \mathcal{T} generates. Hyper-parameters may be viewed as distinguished parameters, often taken from a small number of standard values; for example, the kernel-type used in an SVM, of which only a small set are used in practice, may be seen as a hyper-parameter.

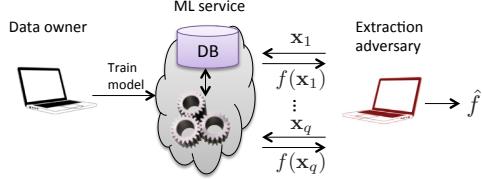


Figure 1: **Diagram of ML model extraction attacks.** A data owner has a model f trained on its data and allows others to make prediction queries. An adversary uses q prediction queries to extract an $\hat{f} \approx f$.

3 Model Extraction Attacks

An ML model extraction attack arises when an adversary obtains black-box access to some *target model* f and attempts to learn a model \hat{f} that closely approximates, or even matches, f (see Figure 1).

As mentioned previously, the restricted case in which f outputs class labels only, matches the membership query setting considered in learning theory, e.g., PAC learning [53] and other previous works [3, 7, 8, 15, 30, 33, 36]. Learning theory algorithms have seen only limited study in practice, e.g., in [36], and our investigation may be viewed as a practice-oriented exploration of this branch of research. Our initial focus, however, is on a different setting common in today’s MLaaS services, which we now explain in detail. Models trained by these services emit data-rich outputs that often include confidence values, and in which partial feature vectors may be considered valid inputs. As we show later, this setting greatly advantages adversaries.

Machine learning services. A number of companies have launched or are planning to launch cloud-based ML services. A common denominator is the ability of users to upload data sets, have the provider run training algorithms on the data, and make the resulting models generally available for prediction queries. Simple-to-use Web APIs handle the entire interaction. This service model lets users capitalize on their data without having to set up their own large-scale ML infrastructure. Details vary greatly across services. We summarize a number of them in Table 2 and now explain some of the salient features.

A model is *white-box* if a user may download a representation suitable for local use. It is *black-box* if accessible only via a prediction query interface. Amazon and Google, for example, provide black-box-only services. Google does not even specify what training algorithm their service uses, while Amazon provides only partial documentation for its feature extraction ex (see Section 5). Some services allow users to monetize trained models by charging others for prediction queries.

To use these services, a user uploads a data set and optionally applies some data pre-processing (e.g., field removal or handling of missing values). She then trains a

Service	White-box	Monetize	Confidence Scores	Logistic Regression	SVM	Neural Network	Decision Tree
Amazon [1]	x	x	✓	✓	x	x	x
Microsoft [38]	x	x	✓	✓	✓	✓	✓
BigML [11]	✓	✓	✓	✓	x	x	✓
PredictionIO [43]	✓	x	x	✓	✓	x	✓
Google [25]	x	✓	✓	✓	✓	✓	✓

Table 2: **Particularities of major MLaaS providers.** ‘White-box’ refers to the ability to download and use a trained model locally, and ‘Monetize’ means that a user may charge other users for black-box access to her models. Model support for each service is obtained from available documentation. The models listed for Google’s API are a projection based on the announced support of models in standard PMML format [25]. Details on ML models are given in Appendix A.

model by either choosing one of many supported model classes (as in BigML, Microsoft, and PredictionIO) or having the service choose an appropriate model class (as in Amazon and Google). Two services have also announced upcoming support for users to upload their own trained models (Google) and their own custom learning algorithms (PredictionIO). When training a model, users may tune various parameters of the model or training-algorithm (e.g., regularizers, tree size, learning rates) and control feature-extraction and transformation methods.

For black-box models, the service provides users with information needed to create and interpret predictions, such as the list of input features and their types. Some services also supply the model class, chosen training parameters, and training data statistics (e.g., BigML gives the range, mean, and standard deviation of each feature).

To get a prediction from a model, a user sends one or more input queries. The services we reviewed accept both synchronous requests and asynchronous ‘batch’ requests for multiple predictions. We further found varying degrees of support for ‘incomplete’ queries, in which some input features are left unspecified [46]. We will show that exploiting incomplete queries can drastically improve the success of some of our attacks. Apart from PredictionIO, all of the services we examined respond to prediction queries with not only class labels, but a variety of additional information, including *confidence scores* (typically class probabilities) for the predicted outputs.

Google and BigML allow model owners to monetize their models by charging other users for predictions. Google sets a minimum price of \$0.50 per 1,000 queries. On BigML, 1,000 queries consume at least 100 *credits*, costing \$0.10–\$5, depending on the user’s subscription.

Attack scenarios. We now describe possible motivations for adversaries to perform model extraction attacks. We then present a more detailed threat model informed by characteristics of the aforementioned ML services.

Avoiding query charges. Successful monetization of

prediction queries by the owner of an ML model f requires confidentiality of f . A malicious user may seek to launch what we call a *cross-user* model extraction attack, stealing f for subsequent free use. More subtly, in black-box-only settings (e.g., Google and Amazon), a service’s business model may involve amortizing up-front training costs by charging users for future predictions. A model extraction attack will undermine the provider’s business model if a malicious user pays less for training and extracting than for paying per-query charges.

Violating training-data privacy. Model extraction could, in turn, leak information about sensitive training data. Prior attacks such as model inversion [4, 23, 24] have shown that access to a model can be abused to infer information about training set points. Many of these attacks work better in white-box settings; model extraction may thus be a stepping stone to such privacy-abusing attacks. Looking ahead, we will see that in some cases, significant information about training data is leaked trivially by successful model extraction, because the model itself directly incorporates training set points.

Stepping stone to evasion. In settings where an ML model serves to detect adversarial behavior, such as identification of spam, malware classification, and network anomaly detection, model extraction can facilitate *evasion attacks*. An adversary may use knowledge of the ML model to avoid detection by it [4, 9, 29, 36, 55].

In all of these settings, there is an inherent assumption of secrecy of the ML model in use. We show that this assumption is broken for all ML APIs that we investigate.

Threat model in detail. Two distinct adversarial models arise in practice. An adversary may be able to make *direct* queries, providing an arbitrary input \mathbf{x} to a model f and obtaining the output $f(\mathbf{x})$. Or the adversary may be able to make only *indirect* queries, i.e., queries on points in input space \mathcal{M} yielding outputs $f(\text{ex}(\mathcal{M}))$. The feature extraction mechanism ex may be unknown to the adversary. In Section 5, we show how ML APIs can further be exploited to “learn” feature extraction mechanisms. Both direct and indirect access to f arise in ML services. (Direct query interfaces arise when clients are expected to perform feature extraction locally.) In either case, the output value can be a class label, a confidence value vector, or some data structure revealing various levels of information, depending on the exposed API.

We model the adversary, denoted by \mathcal{A} , as a randomized algorithm. The adversary’s goal is to use as few queries as possible to f in order to efficiently compute an approximation \hat{f} that closely matches f . We formalize “closely matching” using two different error measures:

- *Test error* R_{test} : This is the average error over a test set D , given by $R_{\text{test}}(f, \hat{f}) = \sum_{(\mathbf{x}, y) \in D} d(f(\mathbf{x}), \hat{f}(\mathbf{x}))/|D|$.

A low test error implies that \hat{f} matches f well for inputs distributed like the training data samples.²

- *Uniform error* R_{unif} : For a set U of vectors uniformly chosen in \mathcal{X} , let $R_{\text{unif}}(f, \hat{f}) = \sum_{\mathbf{x} \in U} d(f(\mathbf{x}), \hat{f}(\mathbf{x}))/|U|$. Thus R_{unif} estimates the fraction of the full feature space on which f and \hat{f} disagree. (In our experiments, we found $|U| = 10,000$ was sufficiently large to obtain stable error estimates for the models we analyzed.)

We define the extraction *accuracy* under test and uniform error as $1 - R_{\text{test}}(f, \hat{f})$ and $1 - R_{\text{unif}}(f, \hat{f})$. Here we implicitly refer to accuracy under 0-1 distance. When assessing how close the class probabilities output by \hat{f} are to those of f (with the total-variation distance) we use the notations $R_{\text{test}}^{\text{TV}}(f, \hat{f})$ and $R_{\text{unif}}^{\text{TV}}(f, \hat{f})$.

An adversary may know any of a number of pieces of information about a target f : What training algorithm \mathcal{T} generated f , the hyper-parameters used with \mathcal{T} , the feature extraction function ex , etc. We will investigate a variety of settings in this work corresponding to different APIs seen in practice. We assume that \mathcal{A} has no more information about a model’s training data, than what is provided by an ML API (e.g., summary statistics). For simplicity, we focus on *proper* model extraction: If \mathcal{A} believes that f belongs to some model class, then \mathcal{A} ’s goal is to extract a model \hat{f} from the *same* class. We discuss some intuition in favor of proper extraction in Appendix D, and leave a broader treatment of *improper* extraction strategies as an interesting open problem.

4 Extraction with Confidence Values

We begin our study of extraction attacks by focusing on prediction APIs that return confidence values. As per Section 2, the output of a query to f thus falls in a range $[0, 1]^c$ where c is the number of classes. To motivate this, we recall that most ML APIs reveal confidence values for models that support them (see Table 2). This includes logistic regressions (LR), neural networks, and decision trees, defined formally in Appendix A. We first introduce a generic *equation-solving* attack that applies to all logistic models (LR and neural networks). In Section 4.2, we present two novel *path-finding* attacks on decision trees.

4.1 Equation-Solving Attacks

Many ML models we consider directly compute class probabilities as a continuous function of the input \mathbf{x} and real-valued model parameters. In this case, an API that reveals these class probabilities provides an adversary \mathcal{A} with samples $(\mathbf{x}, f(\mathbf{x}))$ that can be viewed as equations in the unknown model parameters. For a large class of

²Note that for some D , it is possible that \hat{f} predicts true labels better than f , yet $R_{\text{test}}(f, \hat{f})$ is large, because \hat{f} does not closely match f .

Data set	Synthetic	# records	# classes	# features
Circles	Yes	5,000	2	2
Moons	Yes	5,000	2	2
Blobs	Yes	5,000	3	2
5-Class	Yes	1,000	5	20
Adult (Income)	No	48,842	2	108
Adult (Race)	No	48,842	5	105
Iris	No	150	3	4
Steak Survey	No	331	5	40
GSS Survey	No	16,127	3	101
Digits	No	1,797	10	64
Breast Cancer	No	683	2	10
Mushrooms	No	8,124	2	112
Diabetes	No	768	2	8

Table 3: Data sets used for extraction attacks. We train two models on the Adult data, with targets ‘Income’ and ‘Race’. SVMs and binary logistic regressions are trained on data sets with 2 classes. Multiclass regressions and neural networks are trained on multiclass data sets. For decision trees, we use a set of public models shown in Table 5.

models, these equation systems can be efficiently solved, thus recovering f (or some good approximation of it).

Our approach for evaluating attacks will primarily be experimental. We use a suite of synthetic or publicly available data sets to serve as stand-ins for proprietary data that might be the target of an extraction attack. Table 3 displays the data sets used in this section, which we obtained from various sources: the synthetic ones we generated; the others are taken from public surveys (*Steak Survey* [26] and *GSS Survey* [49]), from *scikit* [42] (*Digits*) or from the UCI ML library [35]. More details about these data sets are in Appendix B.

Before training, we remove rows with missing values, apply *one-hot-encoding* to categorical features, and scale all numeric features to the range $[-1, 1]$. We train our models over a randomly chosen subset of 70% of the data, and keep the rest for evaluation (i.e., to calculate R_{test}). We discuss the impact of different pre-processing and feature extraction steps in Section 5, when we evaluate equation-solving attacks on production ML services.

4.1.1 Binary logistic regression

As a simple starting point, we consider the case of logistic regression (LR). A LR model performs binary classification ($c = 2$), by estimating the probability of a binary response, based on a number of independent features. LR is one of the most popular binary classifiers, due to its simplicity and efficiency. It is widely used in many scientific fields (e.g., medical and social sciences) and is supported by all the ML services we reviewed.

Formally, a LR model is defined by parameters $\mathbf{w} \in \mathbb{R}^d$, $\beta \in \mathbb{R}$, and outputs a probability $f_1(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + \beta)$, where $\sigma(t) = 1/(1 + e^{-t})$. LR is a linear classifier: it defines a *hyperplane* in the feature space \mathcal{X} (defined by $\mathbf{w} \cdot \mathbf{x} + \beta = 0$), that separates the two classes.

Given an oracle sample $(\mathbf{x}, f(\mathbf{x}))$, we get a *linear* equation $\mathbf{w} \cdot \mathbf{x} + \beta = \sigma^{-1}(f_1(\mathbf{x}))$. Thus, $d + 1$ samples are both necessary and sufficient (if the queried \mathbf{x} are linearly independent) to recover \mathbf{w} and β . Note that the required

samples are chosen non-adaptively, and can thus be obtained from a single batch request to the ML service.

We stress that while this extraction attack is rather straightforward, it directly applies, with possibly devastating consequences, to all cloud-based ML services we considered. As an example, recall that some services (e.g., BigML and Google) let model owners monetize black-box access to their models. Any user who wishes to make more than $d + 1$ queries to a model would then minimize the prediction cost by first running a cross-user model extraction attack, and then using the extracted model for personal use, free of charge. As mentioned in Section 3, attackers with a final goal of model-inversion or evasion may also have incentives to first extract the model. Moreover, for services with black-box-only access (e.g., Amazon or Google), a user may abuse the service’s resources to train a model over a large data set D (i.e., $|D| \gg d$), and extract it after only $d + 1$ predictions. Crucially, the extraction cost is independent of $|D|$. This could undermine a service’s business model, should prediction fees be used to amortize the high cost of training.

For each binary data set shown in Table 3, we train a LR model and extract it given $d + 1$ predictions. In all cases, we achieve $R_{\text{test}} = R_{\text{unif}} = 0$. If we compare the probabilities output by f and \hat{f} , $R_{\text{test}}^{\text{TV}}$ and $R_{\text{unif}}^{\text{TV}}$ are lower than 10^{-9} . For these models, the attack requires only 41 queries on average, and 113 at most. On Google’s platform for example, an extraction attack would cost less than \$0.10, and subvert any further model monetization.

4.1.2 Multiclass LRs and Multilayer Perceptrons

We now show that such equation-solving attacks broadly extend to all model classes with a ‘logistic’ layer, including multiclass ($c > 2$) LR and deeper neural networks. We define these models formally in Appendix A.

A multiclass logistic regression (MLR) combines c binary models, each with parameters \mathbf{w}_i, β_i , to form a multiclass model. MLRs are available in all ML services we reviewed. We consider two types of MLR models: softmax and one-vs-rest (OvR), that differ in how the c binary models are trained and combined: A softmax model fits a joint multinomial distribution to all training samples, while a OvR model trains a separate binary LR for each class, and then normalizes the class probabilities.

A MLR model f is defined by parameters $\mathbf{w} \in \mathbb{R}^{cd}$, $\beta \in \mathbb{R}^c$. Each sample $(\mathbf{x}, f(\mathbf{x}))$ gives c equations in \mathbf{w} and β . The equation system is non-linear however, and has no analytic solution. For softmax models for instance, the equations take the form $e^{\mathbf{w}_i \cdot \mathbf{x} + \beta_i} / (\sum_{j=0}^{c-1} e^{\mathbf{w}_j \cdot \mathbf{x} + \beta_j}) = f_i(\mathbf{x})$. A common method for solving such a system is by minimizing an appropriate loss function, such as the logistic loss. With a regularization term, the loss function is *strongly convex*, and the optimization thus con-

Model	Unknowns	Queries	$1 - R_{\text{test}}$	$1 - R_{\text{unif}}$	Time (s)
Softmax	530	265	99.96%	99.75%	2.6
		530	100.00%	100.00%	3.1
OvR	530	265	99.98%	99.98%	2.8
		530	100.00%	100.00%	3.5
MLP	2,225	1,112	98.17%	94.32%	155
		2,225	98.68%	97.23%	168
		4,450	99.89%	99.82%	195
		11,125	99.96%	99.99%	89

Table 4: Success of equation-solving attacks. Models to extract were trained on the Adult data set with multiclass target ‘Race’. For each model, we report the number of unknown model parameters, the number of queries used, and the running time of the equation solver. The attack on the MLP with 11,125 queries converged after 490 epochs.

verges to a *global minimum* (i.e., a function \hat{f} that predicts the same probabilities as f for all available samples). A similar optimization (over class labels rather than probabilities) is actually used for training logistic models. Any MLR implementation can thus easily be adapted for model extraction with equation-solving.

This approach naturally extends to deeper neural networks. We consider multilayer perceptrons (MLP), that first apply a non-linear transform to all inputs (the hidden layer), followed by a softmax regression in the transformed space. MLPs are becoming increasingly popular due to the continued success of deep learning methods; the advent of cloud-based ML services is likely to further boost their adoption. For our attacks, MLPs and MLRs mainly differ in the number of unknowns in the system to solve. For perceptrons with one hidden layer, we have $\mathbf{w} \in \mathbb{R}^{dh+hc}$, $\beta \in \mathbb{R}^{h+c}$, where h is the number of hidden nodes ($h = 20$ in our experiments). Another difference is that the loss function for MLPs is not strongly convex. The optimization may thus converge to a local minimum, i.e., a model \hat{f} that does not exactly match f ’s behavior.

To illustrate our attack’s success, we train a softmax regression, a OvR regression and a MLP on the Adult data set with target ‘Race’ ($c = 5$). For the non-linear equation systems we obtain, we do not know a priori how many samples we need to find a solution (in contrast to linear systems where $d + 1$ samples are necessary and sufficient). We thus explore various query budgets of the form $\alpha \cdot k$, where k is the number of unknown model parameters, and α is a budget scaling factor. For MLRs, we solve the equation system with BFGS [41] in scikit [42]. For MLPs, we use theano [51] to run stochastic gradient descent for 1,000 epochs. Our experiments were performed on a commodity laptop (2-core Intel CPU @3.1GHz, 16GB RAM, no GPU acceleration).

Table 4 shows the extraction success for each model, as we vary α from 0.5 to at most 5. For MLR models (softmax and OvR), the attack is extremely efficient, requiring around one query per unknown parameter of f (each query yields $c = 5$ equations). For MLPs, the system to solve is more complex, with about 4 times more

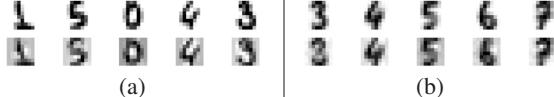


Figure 2: **Training data leakage in KLR models.** (a) Displays 5 of 20 training samples used as representers in a KLR model (top) and 5 of 20 extracted representers (bottom). (b) For a second model, shows the average of all 1,257 representers that the model classifies as a 3, 4, 5, 6 or 7 (top) and 5 of 10 extracted representers (bottom).

unknowns. With a sufficiently over-determined system, we converge to a model \hat{f} that very closely approximates f . As for LR models, queries are chosen non-adaptively, so \mathcal{A} may submit a single ‘batch request’ to the API.

We further evaluated our attacks over all multiclass data sets from Table 3. For MLR models with $k = c \cdot (d + 1)$ parameters (c is the number of classes), k queries were sufficient to achieve perfect extraction ($R_{\text{test}} = R_{\text{unif}} = 0$, $R_{\text{test}}^{\text{TV}}$ and $R_{\text{unif}}^{\text{TV}}$ below 10^{-7}). We use 260 samples on average, and 650 for the largest model (Digits). For MLPs with 20 hidden nodes, we achieved >99.9% accuracy with 5,410 samples on average and 11,125 at most (Adult). With 54,100 queries on average, we extracted a \hat{f} with 100% accuracy over tested inputs. As for binary LRs, we thus find that cross-user model extraction attacks for these model classes can be extremely efficient.

4.1.3 Training Data Leakage for Kernel LR

We now move to a less mainstream model class, *kernel logistic regression* [57], that illustrates how extraction attacks can leak private training data, when a model’s outputs are directly computed as a function of that data.

Kernel methods are commonly used to efficiently extend support vector machines (SVM) to nonlinear classifiers [14], but similar techniques can be applied to logistic regression [57]. Compared to kernel SVMs, kernel logistic regressions (KLR) have the advantage of computing class probabilities, and of naturally extending to multiclass problems. Yet, KLRs have not reached the popularity of kernel SVMs or standard LRs, and are not provided by any MLaaS provider at the time. We note that KLRs could easily be constructed in any ML library that supports both kernel functions and LR models.

A KLR model is a softmax model, where we replace the linear components $\mathbf{w}_i \cdot \mathbf{x} + \beta_i$ by a mapping $\sum_{r=1}^s \alpha_{i,r} K(\mathbf{x}, \mathbf{x}_r) + \beta_i$. Here, K is a kernel function, and the *representers* $\mathbf{x}_1, \dots, \mathbf{x}_s$ are a chosen subset of the training points [57]. More details are in Appendix A.

Each sample $(\mathbf{x}, f(\mathbf{x}))$ from a KLR model yields c equations over the parameters $\boldsymbol{\alpha} \in \mathbb{R}^{sc}$, $\boldsymbol{\beta} \in \mathbb{R}^c$ and the representers $\mathbf{x}_1, \dots, \mathbf{x}_s$. Thus, by querying the model, \mathcal{A} obtains a non-linear equation system, the solution of which leaks training data. This assumes that \mathcal{A} knows the exact number s of representers sampled from the data.

However, we can relax this assumption: First, note that f ’s outputs are unchanged by adding ‘extra’ representers, with weights $\alpha = 0$. Thus, over-estimating s still results in a consistent system of equations, of which a solution is the model f , augmented with unused representers. We will also show experimentally that training data may leak even if \mathcal{A} extracts a model \hat{f} with $s' \ll s$ representers.

We build two KLR models with a *radial-basis function* (RBF) kernel for a data set of handwritten digits. We select 20 random digits as representers for the first model, and all 1,257 training points for the second. We extract the first model, assuming knowledge of s , by solving a system of 50,000 equations in 1,490 unknowns. We use the same approach as for MLPs, i.e., logistic-loss minimization using gradient descent. We initialize the extracted representers to uniformly random vectors in \mathcal{X} , as we assume \mathcal{A} does not know the training data distribution. In Figure 2a, we plot 5 of the model’s representers from the training data, and the 5 closest (in l_1 norm) extracted representers. The attack clearly leaks information on individual training points. We measure the attack’s robustness to uncertainty about s , by attacking the second model with only 10 local representers (10,000 equations in 750 unknowns). Figure 2b shows the *average* image of training points classified as a 3, 4, 5, 6 or 7 by the target model f , along with 5 extracted representers of \hat{f} . Surprisingly maybe, the attack seems to be leaking the ‘average representor’ of each class in the training data.

4.1.4 Model Inversion Attacks on Extracted Models

Access to a model may enable inference of privacy-damaging information, particularly about the training set [4, 23, 24]. The *model inversion attack* explored by Fredrikson et al. [23] uses access to a classifier f to find the input \mathbf{x}_{opt} that maximizes the class probability for class i , i.e., $\mathbf{x}_{\text{opt}} = \text{argmax}_{\mathbf{x} \in \mathcal{X}} f_i(\mathbf{x})$. This was shown to allow recovery of recognizable images of training set members’ faces when f is a facial recognition model.

Their attacks work best in a *white-box* setting, where the attacker knows f and its parameters. Yet, the authors also note that in a black-box setting, remote queries to a prediction API, combined with numerical approximation techniques, enable successful, albeit much less efficient, attacks. Furthermore, their black-box attacks inherently require f to be queried *adaptively*. They leave as an open question making black-box attacks more efficient.

We explore composing an attack that first attempts to *extract* a model $\hat{f} \approx f$, and then uses it with the [23] white-box inversion attack. Our extraction techniques replace adaptive queries with a non-adaptive “batch” query to f , followed by local computation. We show that extraction plus inversion can require fewer queries and less time than performing black-box inversion directly.

As a case study, we use the softmax model from [23], trained over the AT&T Faces data [5]. The data set consists of images of faces (92×112 pixels) of 40 people. The black-box attack from [23] needs about 20,600 queries to reconstruct a recognizable face for a single training set individual. Reconstructing the faces of all 40 individuals would require around 800,000 online queries.

The trained softmax model is much larger than those considered in Section 4.1, with 412,160 unknowns ($d = 10,304$ and $c = 40$). We solve an under-determined system with 41,216 equations (using gradient descent with 200 epochs), and recover a model \hat{f} achieving $R_{\text{test}}^{\text{TV}}, R_{\text{unif}}^{\text{TV}}$ in the order of 10^{-3} . Note that the number of model parameters to extract is linear in the number of people c , whose faces we hope to recover. By using \hat{f} in white-box model inversion attacks, we obtain results that are visually indistinguishable from the ones obtained using the true f . Given the extracted model \hat{f} , we can recover all 40 faces using white-box attacks, incurring around $20 \times$ fewer remote queries to f than with 40 black-box attacks.

For black-box attacks, the authors of [23] estimate a query latency of 70 milliseconds (a little less than in our own measurements of ML services, see Table 1). Thus, it takes 24 minutes to recover a single face (the inversion attack runs in seconds), and 16 hours to recover all 40 images. In contrast, solving the large equation system underlying our model-extraction attack took 10 hours. The 41,216 online queries would take under one hour if executed sequentially and even less with a batch query. The cost of the 40 local white-box attacks is negligible.

Thus, if the goal is to reconstruct faces for all 40 training individuals, performing model inversion over a previously extracted model results in an attack that is both faster and requires $20 \times$ fewer online queries.

4.2 Decision Tree Path-Finding Attacks

Contrary to logistic models, decision trees do not compute class probabilities as a continuous function of their input. Rather, decision trees partition the input space into discrete regions, each of which is assigned a label and confidence score. We propose a new *path-finding* attack, that exploits API particularities to extract the ‘decisions’ taken by a tree when classifying an input.

Prior work on decision tree extraction [7, 12, 33] has focused on trees with Boolean features and outputs. While of theoretical importance, such trees have limited practical use. Kushilevitz and Mansour [33] showed that Boolean trees can be extracted using membership queries (arbitrary queries for class labels), but their algorithm does not extend to more general trees. Here, we propose attacks that exploit ML API specificities, and that apply to decision tree models used in MLaaS platforms.

Our tree model, defined formally in Appendix A, al-

lows for binary and multi-ary splits over categorical features, and binary splits over numeric features. Each leaf of the tree is labeled with a class label and a confidence score. We note that our attacks also apply (often with better results) to *regression trees*. In regression trees, each leaf is labeled with a real-valued output and confidence.

The key idea behind our attack is to use the rich information provided by APIs on a prediction query, as a *pseudo-identifier* for the *path* that the input traversed in the tree. By varying the value of each input feature, we then find the predicates to be satisfied, for an input to follow a given path in the tree. We will also exploit the ability to query *incomplete inputs*, in which each feature x_i is chosen from a space $\mathcal{X}_i \cup \{\perp\}$, where \perp encodes the absence of a value. One way of handling such inputs ([11, 46]) is to label each node in the tree with an output value. On an input, we traverse the tree until we reach a leaf or an internal node with a split over a missing feature, and output that value of that leaf or node.

We formalize these notions by defining *oracles* that \mathcal{A} can query to obtain an identifier for the leaf or internal node reached by an input. In practice, we instantiate these oracles using prediction API peculiarities.

Definition 1 (Identity Oracles). *Let each node v of a tree T be assigned some identifier id_v . A leaf-identity oracle \mathcal{O} takes as input a query $\mathbf{x} \in \mathcal{X}$ and returns the identifier of the leaf of the tree T that is reached on input \mathbf{x} .*

A node-identity oracle \mathcal{O}_{\perp} takes as input a query $\mathbf{x} \in \mathcal{X}_1 \cup \{\perp\} \times \cdots \times \mathcal{X}_d \cup \{\perp\}$ and returns the identifier of the node or leaf of T at which the tree computation halts.

4.2.1 Extraction Algorithms

We now present our path-finding attack (Algorithm 1), that assumes a leaf-identity oracle that returns *unique* identifiers for each leaf. We will relax the uniqueness assumption further on. The attack starts with a random input \mathbf{x} and gets the leaf id from the oracle. We then search for all constraints on \mathbf{x} that have to be satisfied to remain in that leaf, using procedures LINE_SEARCH (for continuous features) and CAT_SPLIT (for categorical features) described below. From this information, we then create new queries for unvisited leaves. Once all leaves have been found, the algorithm returns, for each leaf, the corresponding constraints on \mathbf{x} . We analyze the algorithm’s correctness and complexity in Appendix C.

We illustrate our algorithm with a toy example of a tree over continuous feature *Size* and categorical feature *Color* (see Figure 3). The current query is $\mathbf{x} = \{\text{Size} = 50, \text{Color} = R\}$ and $\mathcal{O}(\mathbf{x}) = \text{id}_2$. Our goal is two-fold: (1) Find the *predicates* that \mathbf{x} has to satisfy to end up in leaf id_2 (i.e., $\text{Size} \in (40, 60]$, $\text{Color} = R$), and (2) create new inputs \mathbf{x}' to explore other paths in the tree.

Algorithm 1 The path-finding algorithm. The notation $\text{id} \leftarrow \mathcal{O}(\mathbf{x})$ means querying the leaf-identity oracle \mathcal{O} with an input \mathbf{x} and obtaining a response id . By $\mathbf{x}[i] \Rightarrow v$ we denote the query \mathbf{x}' obtained from \mathbf{x} by replacing the value of x_i by v .

```

1:  $\mathbf{x}_{\text{init}} \leftarrow \{x_1, \dots, x_d\}$                                 ▷ random initial query
2:  $Q \leftarrow \{\mathbf{x}_{\text{init}}\}$                                          ▷ Set of unprocessed queries
3:  $P \leftarrow \{\}$                                                  ▷ Set of explored leaves with their predicates
4: while  $Q$  not empty do
5:    $\mathbf{x} \leftarrow Q.\text{POP}()$ 
6:    $\text{id} \leftarrow \mathcal{O}(\mathbf{x})$                                      ▷ Call to the leaf identity oracle
7:   if  $\text{id} \in P$  then                                         ▷ Check if leaf already visited
8:     continue
9:   end if
10:  for  $1 \leq i \leq d$  do                                         ▷ Test all features
11:    if IS_CONTINUOUS( $i$ ) then
12:      for  $(\alpha, \beta) \in \text{LINE\_SEARCH}(\mathbf{x}, i, \varepsilon)$  do
13:        if  $x_i \in (\alpha, \beta)$  then
14:           $P[\text{id}].\text{ADD}('x_i \in (\alpha, \beta)')$            ▷ Current interval
15:        else
16:           $Q.\text{PUSH}(\mathbf{x}[i] \Rightarrow \beta)$                   ▷ New leaf to visit
17:        end if
18:      end for
19:    else
20:       $S, V \leftarrow \text{CATEGORY\_SPLIT}(\mathbf{x}, i, \text{id})$ 
21:       $P[\text{id}].\text{ADD}('x_i \in S')$                       ▷ Values for current leaf
22:      for  $v \in V$  do
23:         $Q.\text{PUSH}(\mathbf{x}[i] \Rightarrow v)$                       ▷ New leaves to visit
24:      end for
25:    end if
26:  end for
27: end while

```

The LINE_SEARCH procedure (line 12) tests *continuous features*. We start from bounds on the range of a feature $X_i = [a, b]$. In our example, we have $Size \in [0, 100]$. We set the value of $Size$ in \mathbf{x} to 0 and 100, query \mathcal{O} , and obtain id_1 and id_5 . As the ids do not match, a split on $Size$ occurs on the path to id_2 . With a binary search over feature $Size$ (and all other features in \mathbf{x} fixed), we find all intervals that lead to different leaves, i.e., $[0, 40], (40, 60], (60, 100]$. From these intervals, we find the predicate for the current leaf (i.e., $Size \in (40, 60]$) and build queries to explore new tree paths. To ensure termination of the line search, we specify some *precision* ε . If a split is on a threshold t , we find the value \tilde{t} that is the unique multiple of ε in the range $(t - \varepsilon, t]$. For values x_i with granularity ε , splitting on \tilde{t} is then equivalent to splitting on t .

The CATEGORY_SPLIT procedure (line 20) finds splits on *categorical features*. In our example, we vary the value of *Color* in \mathbf{x} and query \mathcal{O} to get a leaf id for each value. We then build a set S of values that lead to the current leaf, i.e., $S = \{\text{R}\}$, and a set V of values to set in \mathbf{x} to explore other leaves (one representative per leaf). In our example, we could have $V = \{\text{B}, \text{G}, \text{Y}\}$ or $V = \{\text{B}, \text{G}, \text{O}\}$.

Using these two procedures, we thus find the predicates defining the path to leaf id_2 , and generate new queries \mathbf{x}' for unvisited leaves of the tree.

A top-down approach. We propose an empirically more efficient *top-down* algorithm that exploits queries over partial inputs. It extracts the tree ‘layer by layer’,

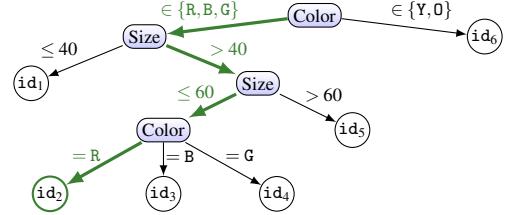


Figure 3: Decision tree over features Color and Size. Shows the path (thick green) to leaf id_2 on input $\mathbf{x} = \{Size = 50, Color = R\}$.

Data set	# records	# classes	# features
IRS Tax Patterns	191,283	51	31
Steak Survey	430	5	12
GSS Survey	51,020	3	7
Email Importance	4,709	2	14
Email Spam	4,601	2	46
German Credit	1,000	2	11
Medical Cover	163,065	$\mathcal{Y} = \mathbb{R}$	13
Bitcoin Price	1,076	$\mathcal{Y} = \mathbb{R}$	7

Table 5: Data sets used for decision tree extraction. Trained trees for these data sets are available in BigML’s public gallery. The last two data sets are used to train regression trees.

starting at the root: We start with an empty query (all features set to \perp) and get the root’s id by querying \mathcal{O}_\perp . We then set each feature in turn and query \mathcal{O} again. For exactly one feature (the root’s splitting feature), the input will reach a different node. With similar procedures as described previously, we extract the root’s splitting criterion, and recursively search lower layers of the tree.

Duplicate identities. As we verify empirically, our attacks are resilient to some nodes or leaves sharing the same id . We can modify line 7 in Algorithm 1 to detect id duplicates, by checking not only whether a leaf with the current id was already visited, but also whether the current query violates that leaf’s predicates. The main issue with duplicate ids comes from the LINE_SEARCH and CATEGORY_SPLIT procedures: if two queries \mathbf{x} and \mathbf{x}' differ in a single feature and reach different leaves with the same id , the split on that feature will be missed.

4.2.2 Attack Evaluation

Our tree model (see Appendix A) is the one used by BigML. Other ML services use similar tree models. For our experiments, we downloaded eight public decision trees from BigML (see Table 5), and queried them locally using available API bindings. More details on these models are in Appendix B. We show *online* extraction attacks on black-box models from BigML in Section 5.

To emulate black-box model access, we first issue online queries to BigML, to determine the information contained in the service’s responses. We then simulate black-box access locally, by discarding any extra information returned by the local API. Specifically, we make use of the following fields in query responses:

Model	Leaves	Unique IDs	Depth	Without incomplete queries			With incomplete queries		
				$1 - R_{\text{test}}$	$1 - R_{\text{unif}}$	Queries	$1 - R_{\text{test}}$	$1 - R_{\text{unif}}$	Queries
IRS Tax Patterns	318	318	8	100.00%	100.00%	101,057	100.00%	100.00%	29,609
Steak Survey	193	28	17	92.45%	86.40%	3,652	100.00%	100.00%	4,013
GSS Survey	159	113	8	99.98%	99.61%	7,434	100.00%	99.65%	2,752
Email Importance	109	55	17	99.13%	99.90%	12,888	99.81%	99.99%	4,081
Email Spam	219	78	29	87.20%	100.00%	42,324	99.70%	100.00%	21,808
German Credit	26	25	11	100.00%	100.00%	1,722	100.00%	100.00%	1,150
Medical Cover	49	49	11	100.00%	100.00%	5,966	100.00%	100.00%	1,788
Bitcoin Price	155	155	9	100.00%	100.00%	31,956	100.00%	100.00%	7,390

Table 6: Performance of extraction attacks on public models from BigML. For each model, we report the number of leaves in the tree, the number of unique identifiers for those leaves, and the maximal tree depth. The chosen granularity ϵ for continuous features is 10^{-3} .

- **Prediction.** This entry contains the predicted class label (classification) or real-valued output (regression).
- **Confidence.** For classification and regression trees, BigML computes confidence scores based on a confidence interval for predictions at each node [11]. The prediction and confidence value constitute a node’s id.
- **Fields.** Responses to black-box queries contain a ‘fields’ property, that lists all features that appear either in the input query or on the path traversed in the tree. If a partial query x reaches an internal node v , this entry tells us which feature v splits on (the feature is in the ‘fields’ entry, but not in the input x). We make use of this property for the top-down attack variant.

Table 6 displays the results of our attacks. For each tree, we give its number of leaves, the number of unique leaf ids, and the tree depth. We display the success rate for Algorithm 1 and for the “top-down” variant with incomplete queries. Querying partial inputs vastly improves our attack: we require far less queries (except for the Steak Survey model, where Algorithm 1 only visits a fraction of all leaves and thus achieves low success) and achieve higher accuracy for trees with duplicate leaf ids. As expected, both attacks achieve perfect extraction when all leaves have unique ids. While this is not always the case for classification trees, it is far more likely for regression trees, where both the label and confidence score take real values. Surprisingly maybe, the top-down approach also fully extracts some trees with a large number of duplicate leaf ids. The attacks are also efficient: The top-down approach takes less than 10 seconds to extract a tree, and Algorithm 1 takes less than 6 minutes for the largest tree. For online attacks on ML services, discussed next, this cost is trumped by the delay for the inherently adaptive prediction queries that are issued.

5 Online Model Extraction Attacks

In this section, we showcase *online* model extraction attacks against two ML services: BigML and Amazon. For BigML, we focus on extracting models set up by a user, who wishes to charge for predictions. For Amazon, our goal is to extract a model trained by ourselves, to which we only get black-box access. Our attacks only use ex-

Model	OHE	Binning	Queries	Time (s)	Price (\$)
Circles	-	Yes	278	28	0.03
Digits	-	No	650	70	0.07
Iris	-	Yes	644	68	0.07
Adult	Yes	Yes	1,485	149	0.15

Table 7: Results of model extraction attacks on Amazon. OHE stands for one-hot-encoding. The reported query count is the number used to find quantile bins (at a granularity of 10^{-3}), plus those queries used for equation-solving. Amazon charges \$0.0001 per prediction [1].

posed APIs, and do not in any way attempt to bypass the services’ authentication or access-control mechanisms. We only attack models trained in our own accounts.

5.1 Case Study 1: BigML

BigML currently only allows monetization of decision trees [11]. We train a tree on the *German Credit* data, and set it up as a black-box model. The tree has 26 leaves, two of which share the same label and confidence score. From another account, we extract the model using the two attacks from Section 4.2. We first find the tree’s number of features, their type and their range, from BigML’s public gallery. Our attacks (Algorithm 1 and the top-down variant) extract an exact description of the tree’s paths, using respectively 1,722 and 1,150 queries. Both attacks’ duration (1,030 seconds and 631 seconds) is dominated by query latency (≈ 500 ms/query). The monetary cost of the attack depends on the per-prediction-fee set by the model owner. In any case, a user who wishes to make more than 1,150 predictions has economic incentives to run an extraction attack.

5.2 Case Study 2: Amazon Web Services

Amazon uses logistic regression for classification, and provides black-box-only access to trained models [1]. By default, Amazon uses two feature extraction techniques: (1) Categorical features are *one-hot-encoded*, i.e., the input space $\mathcal{M}_i = \mathbb{Z}_k$ is mapped to k binary features encoding the input value. (2) *Quantile binning* is used for numeric features. The training data values are split into k -quantiles (k equally-sized bins), and the input space $\mathcal{M}_i = [a, b]$ is mapped to k binary features encoding the bin that a value falls into. Note that $|\mathcal{X}| > |\mathcal{M}|$,

i.e., ex increases the number of features. If \mathcal{A} reverse-engineers ex , she can query the service on samples M in input space, compute $\mathbf{x} = \text{ex}(M)$ locally, and extract f in feature-space using equation-solving.

We apply this approach to models trained by Amazon. Our results are summarized in Table 7. We first train a model with no categorical features, and quantile binning disabled (this is a manually tunable parameter), over the Digits data set. The attack is then identical to the one considered in Section 4.1.2: using 650 queries to Amazon, we extract a model that achieves $R_{\text{test}} = R_{\text{unif}} = 0$.

We now consider models with feature extraction enabled. We assume that \mathcal{A} knows the input space \mathcal{M} , but not the training data distribution. For one-hot-encoding, knowledge of \mathcal{M} suffices to apply the same encoding locally. For quantile binning however, applying ex locally requires knowledge of the training data quantiles. To reverse-engineer the binning transformation, we use line-searches similar to those we used for decision trees: For each numeric feature, we search the feature’s range in input space for thresholds (up to a granularity ϵ) where f ’s output changes. This indicates our value landed in an adjacent bin, with a different learned regression coefficient. Note that learning the bin boundaries may be interesting in its own right, as it leaks information about the training data distribution. Having found the bin boundaries, we can apply both one-hot-encoding and binning locally, and extract f over its feature space. As we are restricted to queries over \mathcal{M} , we cannot define an arbitrary system of equations over \mathcal{X} . Building a well-determined and consistent system can be difficult, as the encoding ex generates sparse inputs over \mathcal{X} . However, Amazon facilitates this process with the way it handles queries with *missing features*: if a feature is omitted from a query, all corresponding features in \mathcal{X} are set to 0. For a linear model for instance, we can trivially re-construct the model by issuing queries with a single feature specified, such as to obtain equations with a single unknown in \mathcal{X} .

We trained models for the Circles, Iris and Adult data sets, with Amazon’s default feature-extraction settings. Table 7 shows the results of our attacks, for the reverse-engineering of ex and extraction of f . For binary models (Circles and Adult), we use $d + 1$ queries to solve a linear equation-system over \mathcal{X} . For models with $c > 2$ classes, we use $c \cdot (d + 1)$ queries. In all cases, the extracted model matches f on 100% of tested inputs. To optimize the query complexity, the queries we use to find quantile bins are re-used for equation-solving. As line searches require adaptive queries, we do not use batch predictions. However, even for the Digits model, we resorted to using real-time predictions, because of the service’s significant overhead in evaluating batches. For attacks that require a large number of non-adaptive queries, we expect batch predictions to be faster than real-time predictions.

5.3 Discussion

Additional feature extractors. In some ML services we considered, users may enable further feature extractors. A common transformation is feature scaling or normalization. If \mathcal{A} has access to training data statistics (as provided by BigML for instance), applying the transformation locally is trivial. More generally, for models with a linear input layer (i.e., logistic regressions, linear SVMs, MLPs) the scaling or normalization can be seen as being applied to the learned weights, rather than the input features. We can thus view the composition $f \circ \text{ex}$ as a model f' that operates over the ‘un-scaled’ input space \mathcal{M} and extract f' directly using equation-solving.

Further extractors include text analysis (e.g., bag-of-words or n-gram models) and Cartesian products (grouping many features into one). We have not analyzed these in this work, but we believe that they could also be easily reverse-engineered, especially given some training data statistics and the ability to make incomplete queries.

Learning unknown model classes or hyper-parameters. For our online attacks, we obtained information about the model class of f , the enabled feature extraction ex , and other hyper-parameters, directly from the ML service or its documentation. More generally, if \mathcal{A} does not have full certainty about certain model characteristics, it may be able to narrow down a guess to a small range. Model hyper-parameters for instance (such as the free parameter of an RBF kernel) are typically chosen through cross-validation over a default range of values.

Given a set of attack strategies with varying assumptions, \mathcal{A} can use a generic *extract-and-test* approach: each attack is applied in turn, and evaluated by computing R_{test} or R_{unif} over a chosen set of points. The adversary succeeds if any of the strategies achieves a low error. Note that \mathcal{A} needs to interact with the model f only once, to obtain responses for a chosen set of extraction samples and test samples, that can be re-used for each strategy.

Our attacks on Amazon’s service followed this approach: We first formulated guesses for model characteristics left unspecified by the documentation (e.g., we found no mention of one-hot-encoding, or of how missing inputs are handled). We then evaluated our assumptions with successive extraction attempts. Our results indicate that Amazon uses softmax regression and does not create binary predictors for missing values. Interestingly, BigML takes the ‘opposite’ approach (i.e., BigML uses OvR regression and adds predictors for missing values).

6 Extraction Given Class Labels Only

The successful attacks given in Sections 4 and 5 show the danger of revealing confidence values. While current

ML services have been designed to reveal rich information, our attacks may suggest that returning only labels would be safer. Here we explore model extraction in a setting with no confidence scores. We will discuss further countermeasures in Section 7. We primarily focus on settings where \mathcal{A} can make *direct* queries to an API, i.e., queries for arbitrary inputs $\mathbf{x} \in \mathcal{X}$. We briefly discuss *indirect* queries in the context of linear classifiers.

The Lowd-Meek attack. We start with the prior work of Lowd and Meek [36]. They present an attack on any linear classifier, assuming black-box oracle access with membership queries that return just the predicted class label. A linear classifier is defined by a vector $\mathbf{w} \in \mathbb{R}^d$ and a constant $\beta \in \mathbb{R}$, and classifies an instance \mathbf{x} as *positive* if $\mathbf{w} \cdot \mathbf{x} + \beta > 0$ and *negative* otherwise. SVMs with linear kernels and binary LRs are examples of linear classifiers. Their attack uses line searches to find points arbitrarily close to f 's decision boundary (points for which $\mathbf{w} \cdot \mathbf{x} + \beta \approx 0$), and extracts \mathbf{w} and β from these samples.

This attack only works for linear binary models. We describe a straightforward extension to some non-linear models, such as polynomial kernel SVMs. Extracting a polynomial kernel SVM can be reduced to extracting a linear SVM in the transformed feature space. Indeed, for any kernel $K_{\text{poly}}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \cdot \mathbf{x}' + 1)^d$, we can derive a projection function $\phi(\cdot)$, so that $K_{\text{poly}}(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \cdot \phi(\mathbf{x}')$. This transforms the kernel SVM into a linear one, since the decision boundary now becomes $\mathbf{w}^F \cdot \phi(\mathbf{x}) + \beta = 0$ where $\mathbf{w}^F = \sum_{i=1}^t \alpha_i \phi(\mathbf{x}_i)$. We can use the Lowd-Meek attack to extract \mathbf{w}^F and β as long as $\phi(\mathbf{x})$ and its inverse are feasible to compute; this is unfortunately not the case for the more common RBF kernels.³

The retraining approach. In addition to evaluating the Lowd-Meek attack against ML APIs, we introduce a number of other approaches based on the broad strategy of re-training a model locally, given input-output examples. Informally, our hope is that by extracting a model that achieves low *training error* over the queried samples, we would effectively approximate the target model's decision boundaries. We consider three re-training strategies, described below. We apply these to the model classes that we previously extracted using equation-solving attacks, as well as to SVMs.⁴

- (1) **Retraining with uniform queries.** This baseline strategy simply consists in sampling m points $\mathbf{x}_i \in \mathcal{X}$ uniformly at random, querying the oracle, and training a model \hat{f} on these samples.

³We did explore using approximations of ϕ , but found that the adaptive re-training techniques discussed in this section perform better.

⁴We do not expect retraining attacks to work well for decision trees, because of the greedy approach taken by learning algorithms. We have not evaluated extraction of trees, given class labels only, in this work.

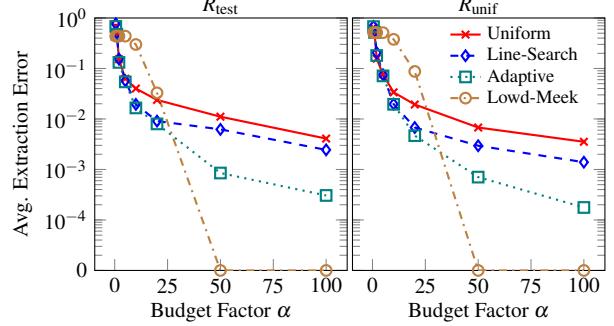


Figure 4: **Average error of extracted linear models.** Results are for different extraction strategies applied to models trained on all binary data sets from Table 3. The left shows R_{test} and the right shows R_{unif} .

- (2) **Line-search retraining.** This strategy can be seen as a model-agnostic generalization of the Lowd-Meek attack. It issues m adaptive queries to the oracle using line search techniques, to find samples close to the decision boundaries of f . A model \hat{f} is then trained on the m queried samples.
- (3) **Adaptive retraining.** This strategy applies techniques from active learning [18, 47]. For some number r of rounds and a query budget m , it first queries the oracle on $\frac{m}{r}$ uniform points, and trains a model \hat{f} . Over a total of r rounds, it then selects $\frac{m}{r}$ new points, along the decision boundary of \hat{f} (intuitively, these are points \hat{f} is *least certain* about), and sends those to the oracle before retraining \hat{f} .

6.1 Linear Binary Models

We first explore how well the various approaches work in settings where the Lowd-Meek attack can be applied. We evaluate their attack and our three retraining strategies for logistic regression models trained over the binary data sets shown in Table 3. These models have $d+1$ parameters, and we vary the query budget as $\alpha \cdot (d+1)$, for $0.5 \leq \alpha \leq 100$. Figure 4 displays the average errors R_{test} and R_{unif} over all models, as a function of α .

The retraining strategies that search for points near the decision boundary clearly perform better than simple uniform retraining. The adaptive strategy is the most efficient of our three strategies. For relatively low budgets, it even outperforms the Lowd-Meek attack. However, for budgets large enough to run line searches in each dimension, the Lowd-Meek attack is clearly the most efficient.

For the models we trained, about 2,050 queries on average, and 5,650 at most, are needed to run the Lowd-Meek attack effectively. This is 50× more queries than what we needed for equation-solving attacks. With 827 queries on average, adaptive retraining yields a model \hat{f} that matches f on over 99% of tested inputs. Thus, even if an ML API only provides class labels, efficient extrac-

tion attacks on linear models remain possible.

We further consider a setting where feature-extraction (specifically one-hot-encoding of categorical features) is applied by the ML service, rather than by the user. \mathcal{A} is then limited to indirect queries in input space. Lowd and Meek [36] note that their extraction attack does not work in this setting, as \mathcal{A} can not run line searches directly over \mathcal{X} . In contrast, for the linear models we trained, we observed no major difference in extraction accuracy for the adaptive-retraining strategy, when limited to queries over \mathcal{M} . We leave an in-depth study of model extraction with indirect queries, and class labels only, for future work.

6.2 Multiclass LR Models

The Lowd-Meek attack is not applicable in multiclass ($c > 2$) settings, even when the decision boundary is a combination of linear boundaries (as in multiclass regression) [39, 50]. We thus focus on evaluating the three retraining attacks we introduced, for the type of ML models we expect to find in real-world applications.

We focus on softmax models here, as softmax and one-vs-rest models have identical output behaviors when only class labels are provided: in both cases, the class label for an input \mathbf{x} is given by $\text{argmax}_i(\mathbf{w}_i \cdot \mathbf{x} + \beta_i)$. From an extractor’s perspective, it is thus irrelevant whether the target was trained using a softmax or OvR approach.

We evaluate our attacks on softmax models trained on the multiclass data sets shown in Table 3. We again vary the query budget as a factor α of the number of model parameters, namely $\alpha \cdot c \cdot (d + 1)$. Results are displayed in Figure 5. We observe that the adaptive strategy clearly performs best and that the line-search strategy does not improve over uniform retraining, possibly because the line-searches have to be split across multiple decision-boundaries. We further note that all strategies achieve lower R_{test} than R_{unif} . It thus appears that for the models we trained, points from the test set are on average ‘far’ from the decision boundaries of f (i.e., the trained models separate the different classes with large margins).

For all models, $100 \cdot c \cdot (d + 1)$ queries resulted in extraction accuracy above 99.9%. This represents 26,000 queries on average, and 65,000 at the most (Digits data set). Our equation-solving attacks achieved similar or better results with $100 \times$ less queries. Yet, for scenarios with high monetary incentives (e.g., intrusion detector evasion), extraction attacks on MLR models may be attractive, even if APIs only provide class labels.

6.3 Neural Networks

We now turn to attacks on more complex deep neural networks. We expect these to be harder to retrain than multiclass regressions, as deep networks have more pa-

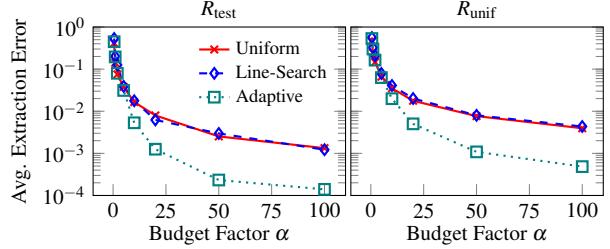


Figure 5: Average error of extracted softmax models. Results are for three retraining strategies applied to models trained on all multiclass data sets from Table 3. The left shows R_{test} and the right shows R_{unif} .

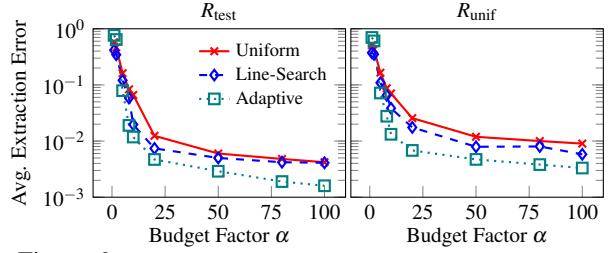


Figure 6: Average error of extracted RBF kernel SVMs. Results are for three retraining strategies applied to models trained on all binary data sets from Table 3. The left shows R_{test} and the right shows R_{unif} .

rameters and non-linear decision-boundaries. Therefore, we may need to find a large number of points close to a decision boundary in order to extract it accurately.

We evaluated our attacks on the multiclass models from Table 3. For the tested query budgets, line-search and adaptive retraining gave little benefit over uniform retraining. For a budget of $100 \cdot k$, where k is the number of model parameters, we get $R_{\text{test}} = 99.16\%$ and $R_{\text{unif}} = 98.24\%$, using 108,200 queries per model on average. Our attacks might improve for higher budgets but it is unclear whether they would then provide any monetary advantage over using ML APIs in an honest way.

6.4 RBF Kernel SVMs

Another class of nonlinear models that we consider are support-vector machines (SVMs) with radial-basis function (RBF) kernels. A kernel SVM first maps inputs into a higher-dimensional space, and then finds the hyperplane that maximally separates the two classes. As mentioned in Section 6, SVMs with polynomial kernels can be extracted using the Lowd-Meek attack in the transformed feature space. For RBF kernels, this is not possible because the transformed space has infinite dimension.

SVMs do not provide class probability estimates. Our only applicable attack is thus retraining. As for linear models, we vary the query budget as $\alpha \cdot (d + 1)$, where d is the input dimension. We further use the *extract-and-test* approach from Section 5 to find the value of the RBF kernel’s *hyper-parameter*. Results of our attacks are in

Figure 6. Again, we see that adaptive retraining performs best, even though the decision boundary to extract is non-linear (in input space) here. Kernel SVMs models are overall harder to retrain than models with linear decision boundaries. Yet, for our largest budgets (2,050 queries on average), we do extract models with over 99% accuracy, which may suffice in certain adversarial settings.

7 Extraction Countermeasures

We have shown in Sections 4 and 5 that adversarial clients can effectively extract ML models given access to rich prediction APIs. Given that this undermines the financial models targeted by some ML cloud services, and potentially leaks confidential training data, we believe researchers should seek countermeasures.

In Section 6, we analyzed the most obvious defense against our attacks: prediction API minimization. The constraint here is that the resulting API must still be useful in (honest) applications. For example, it is simple to change APIs to not return confidences and not respond to incomplete queries, assuming applications can get by without it. This will prevent many of our attacks, most notably the ones described in Section 4 as well as the feature discovery techniques used in our Amazon case study (Section 5). Yet, we showed that even if we strip an API to only provide class labels, successful attacks remain possible (Section 6), albeit at a much higher query cost.

We discuss further potential countermeasures below.

Rounding confidences. Applications might need confidences, but only at lower granularity. A possible defense is to round confidence scores to some fixed precision [23]. We note that ML APIs already work with some finite precision when answering queries. For instance, BigML reports confidences with 5 decimal places, and Amazon provides values with 16 significant digits.

To understand the effects of limiting precision further, we re-evaluate equation-solving and decision tree path-finding attacks with confidence scores rounded to a fixed decimal place. For equation-solving attacks, rounding the class probabilities means that the solution to the obtained equation-system might not be the target f , but some truncated version of it. For decision trees, rounding confidence scores increases the chance of node id collisions, and thus decreases our attacks’ success rate.

Figure 7 shows the results of experiments on softmax models, with class probabilities rounded to 2–5 decimals. We plot only R_{test} , the results for R_{unif} being similar. We observe that class probabilities rounded to 4 or 5 decimal places (as done already in BigML) have no effect on the attack’s success. When rounding further to 3 and 2 decimal places, the attack is weakened, but still vastly outperforms adaptive retraining using class labels only.

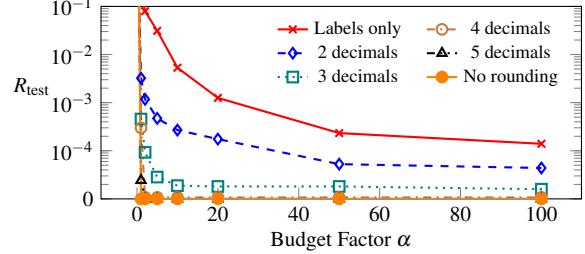


Figure 7: **Effect of rounding on model extraction.** Shows the average test error of equation-solving attacks on softmax models trained on the benchmark suite (Table 3), as we vary the number of significant digits in reported class probabilities. Extraction with no rounding and with class labels only (adaptive retraining) are added for comparison.

For regression trees, rounding has no effect on our attacks. Indeed, for the models we considered, the output itself is unique in each leaf (we could also round outputs, but the impact on utility may be more critical). For classification trees, we re-evaluated our top-down attack, with confidence scores rounded to fewer than 5 decimal places. The attacks on the ‘IRS Tax Patterns’ and ‘Email Importance’ models are the most resilient, and suffer no success degradation before scores are rounded to 2 decimal places. For the other models, rounding confidences to 3 or 4 decimal places severely undermines our attack.

Differential privacy. Differential privacy (DP) [22] and its variants [34] have been explored as mechanisms for protecting, in particular, the privacy of ML training data [54]. DP learning has been applied to regressions [17, 56], SVMs [44], decision trees [31] and neural networks [48]. As some of our extraction attacks leak training data information (Section 4.1.3), one may ask whether DP can prevent extraction, or at least reduce the severity of the privacy violations that extraction enables.

Consider naive application of DP to protect individual training data elements. This should, in theory, decrease the ability of an adversary \mathcal{A} to learn information about training set elements, when given access to prediction queries. One would not expect, however, that this prevents model extraction, as DP is not defined to do so: consider a trivially useless learning algorithm for binary logistic regression, that discards the training data and sets \mathbf{w} and β to 0. This algorithm is differentially private, yet \mathbf{w} and β can easily be recovered using equation-solving.

A more appropriate strategy would be to apply DP directly to the model parameters, which would amount to saying that a query should not allow \mathcal{A} to distinguish between closely neighboring model parameters. How exactly this would work and what privacy budgets would be required is left as an open question by our work.

Ensemble methods. Ensemble methods such as random forests return as prediction an aggregation of pre-

dictions by a number of individual models. While we have not experimented with ensemble methods as targets, we suspect that they may be more resilient to extraction attacks, in the sense that attackers will only be able to obtain relatively coarse approximations of the target function. Nevertheless, ensemble methods may still be vulnerable to other attacks such as model evasion [55].

8 Related Work

Our work is related to the extensive literature on learning theory, such as PAC learning [53] and its variants [3, 8]. Indeed, extraction can be viewed as a type of learning, in which an unknown instance of a known hypothesis class (model type) is providing labels (without error). This is often called learning with membership queries [3]. Our setting differs from these in two ways. The first is conceptual: in PAC learning one builds algorithms to learn a concept — the terminology belies the motivation of formalizing learning from data. In model extraction, an attacker is literally given a function oracle that it seeks to illicitly determine. The second difference is more pragmatic: prediction APIs reveal richer information than assumed in prior learning theory work, and we exploit that.

Algorithms for learning with membership queries have been proposed for Boolean functions [7, 15, 30, 33] and various binary classifiers [36, 39, 50]. The latter line of work, initiated by Lowd and Meek [36], studies strategies for model evasion, in the context of spam or fraud detectors [9, 29, 36, 37, 55]. Intuitively, model extraction seems harder than evasion, and this is corroborated by results from theory [36, 39, 50] and practice [36, 55].

Evasion attacks fall into the larger field of *adversarial machine learning*, that studies machine learning in general adversarial settings [6, 29]. In that context, a number of authors have considered strategies and defenses for *poisoning* attacks, that consist in injecting maliciously crafted samples into a model’s train or test data, so as to decrease the learned model’s accuracy [10, 21, 32, 40, 45].

In a non-malicious setting, improper model extraction techniques have been applied for interpreting [2, 19, 52] and compressing [16, 27] complex neural networks.

9 Conclusion

We demonstrated how the flexible prediction APIs exposed by current ML-as-a-service providers enable new model extraction attacks that could subvert model monetization, violate training-data privacy, and facilitate model evasion. Through local experiments and online attacks on two major providers, BigML and Amazon, we illustrated the efficiency and broad applicability of attacks that exploit common API features, such as the

availability of confidence scores or the ability to query arbitrary partial inputs. We presented a generic *equation-solving* attack for models with a logistic output layer and a novel *path-finding* algorithm for decision trees.

We further explored potential countermeasures to these attacks, the most obvious being a restriction on the information provided by ML APIs. Building upon prior work from learning-theory, we showed how an attacker that only obtains class labels for adaptively chosen inputs, may launch less effective, yet potentially harmful, *retraining attacks*. Evaluating these attacks, as well as more refined countermeasures, on production-grade ML services is an interesting avenue for future work.

Acknowledgments. We thank Martín Abadi and the anonymous reviewers for their comments. This work was supported by NSF grants 1330599, 1330308, and 1546033, as well as a generous gift from Microsoft.

References

- [1] AMAZON WEB SERVICES. <https://aws.amazon.com/machine-learning>. Accessed Feb. 10, 2016.
- [2] ANDREWS, R., DIEDERICH, J., AND TICKLE, A. Survey and critique of techniques for extracting rules from trained artificial neural networks. *KBS* 8, 6 (1995), 373–389.
- [3] ANGLUIN, D. Queries and concept learning. *Machine learning* 2, 4 (1988), 319–342.
- [4] ATENIESE, G., MANCINI, L. V., SPOGNARDI, A., VILLANI, A., VITALI, D., AND FELICI, G. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *IJSN* 10, 3 (2015), 137–150.
- [5] AT&T LABORATORIES CAMBRIDGE. The ORL database of faces. <http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html>.
- [6] BARRENO, M., NELSON, B., SEARS, R., JOSEPH, A. D., AND TYGAR, J. D. Can machine learning be secure? In *ASIACCS* (2006), ACM, pp. 16–25.
- [7] BELLARE, M. A technique for upper bounding the spectral norm with applications to learning. In *COLT* (1992), ACM, pp. 62–70.
- [8] BENEDEK, G. M., AND ITAI, A. Learnability with respect to fixed distributions. *TCS* 86, 2 (1991), 377–389.
- [9] BIGGIO, B., CORONA, I., MAIORCA, D., NELSON, B., ŠRNDIĆ, N., LASKOV, P., GIACINTO, G., AND ROLI, F. Evasion attacks against machine learning at test time. In *ECML PKDD*. Springer, 2013, pp. 387–402.
- [10] BIGGIO, B., NELSON, B., AND LASKOV, P. Poisoning attacks against support vector machines. In *ICML* (2012).
- [11] BIGML. <https://www.bigml.com>. Accessed Feb. 10, 2016.
- [12] BLUM, A. L., AND LANGLEY, P. Selection of relevant features and examples in machine learning. *Artificial intelligence* 97, 1 (1997), 245–271.
- [13] BLUMER, A., EHRENFEUCHT, A., HAUSLER, D., AND WARMUTH, M. K. Occam’s razor. *Readings in machine learning* (1990), 201–204.
- [14] BOSER, B. E., GUYON, I. M., AND VAPNIK, V. N. A training algorithm for optimal margin classifiers. In *COLT* (1992), ACM, pp. 144–152.

- [15] BSHOUTY, N. H. Exact learning boolean functions via the monotone theory. *Inform. Comp.* 123, 1 (1995), 146–153.
- [16] BUCILUĀ, C., CARUANA, R., AND NICULESCU-MIZIL, A. Model compression. In *KDD* (2006), ACM, pp. 535–541.
- [17] CHAUDHURI, K., AND MONTELEONI, C. Privacy-preserving logistic regression. In *NIPS* (2009), pp. 289–296.
- [18] COHN, D., ATLAS, L., AND LADNER, R. Improving generalization with active learning. *Machine learning* 15, 2 (1994), 201–221.
- [19] CRAVEN, M. W., AND SHAVLIK, J. W. Extracting tree-structured representations of trained networks. In *NIPS* (1996).
- [20] CYBENKO, G. Approximation by superpositions of a sigmoidal function. *MCSS* 2, 4 (1989), 303–314.
- [21] DALVI, N., DOMINGOS, P., SANGHAI, S., VERMA, D., ET AL. Adversarial classification. In *KDD* (2004), ACM, pp. 99–108.
- [22] DWORK, C. Differential privacy. In *ICALP* (2006), Springer.
- [23] FREDRIKSON, M., JHA, S., AND RISTENPART, T. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS* (2015), ACM, pp. 1322–1333.
- [24] FREDRIKSON, M., LANTZ, E., JHA, S., LIN, S., PAGE, D., AND RISTENPART, T. Privacy in pharmacogenetics: An end-to-end case study of personalized Warfarin dosing. In *USENIX Security* (2014), pp. 17–32.
- [25] GOOGLE PREDICTION API. <https://cloud.google.com/prediction>. Accessed Feb. 10, 2016.
- [26] HICKEY, W. How Americans Like their Steak. <http://fivethirtyeight.com/datalab/how-americans-like-their-steak>, 2014. Accessed Feb. 10, 2016.
- [27] HINTON, G., VINYALS, O., AND DEAN, J. Distilling the knowledge in a neural network. *arXiv:1503.02531* (2015).
- [28] HORNIK, K., STINCHCOMBE, M., AND WHITE, H. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.
- [29] HUANG, L., JOSEPH, A. D., NELSON, B., RUBINSTEIN, B. I., AND TYGAR, J. Adversarial machine learning. In *AISec* (2011), ACM, pp. 43–58.
- [30] JACKSON, J. An efficient membership-query algorithm for learning DNF with respect to the uniform distribution. In *FOCS* (1994), IEEE, pp. 42–53.
- [31] JAGANNATHAN, G., PILLAIPAKKAMNATT, K., AND WRIGHT, R. N. A practical differentially private random decision tree classifier. In *ICDMW* (2009), IEEE, pp. 114–121.
- [32] KLOFT, M., AND LASKOV, P. Online anomaly detection under adversarial impact. In *AISTATS* (2010), pp. 405–412.
- [33] KUSHLEVITZ, E., AND MANSOUR, Y. Learning decision trees using the Fourier spectrum. *SICOMP* 22, 6 (1993), 1331–1348.
- [34] LI, N., QARDAJI, W., SU, D., WU, Y., AND YANG, W. Membership privacy: A unifying framework for privacy definitions. In *CCS* (2013), ACM.
- [35] LICHMAN, M. UCI machine learning repository, 2013.
- [36] LOWD, D., AND MEEK, C. Adversarial learning. In *KDD* (2005), ACM, pp. 641–647.
- [37] LOWD, D., AND MEEK, C. Good word attacks on statistical spam filters. In *CEAS* (2005).
- [38] MICROSOFT AZURE. <https://azure.microsoft.com/services/machine-learning>. Accessed Feb. 10, 2016.
- [39] NELSON, B., RUBINSTEIN, B. I., HUANG, L., JOSEPH, A. D., LEE, S. J., RAO, S., AND TYGAR, J. Query strategies for evading convex-inducing classifiers. *JMLR* 13, 1 (2012), 1293–1332.
- [40] NEWSOME, J., KARP, B., AND SONG, D. Paragraph: Thwarting signature learning by training maliciously. In *RAID* (2006), Springer, pp. 81–105.
- [41] NOCEDAL, J., AND WRIGHT, S. *Numerical optimization*. Springer Science & Business Media, 2006.
- [42] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PAS-SOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. Scikit-learn: Machine learning in Python. *JMLR* 12 (2011), 2825–2830.
- [43] PREDICTIONIO. <http://prediction.io>. Accessed Feb. 10, 2016.
- [44] RUBINSTEIN, B. I., BARTLETT, P. L., HUANG, L., AND TAFT, N. Learning in a large function space: Privacy-preserving mechanisms for SVM learning. *JPC* 4, 1 (2012), 4.
- [45] RUBINSTEIN, B. I., NELSON, B., HUANG, L., JOSEPH, A. D., LAU, S.-H., RAO, S., TAFT, N., AND TYGAR, J. Antidote: understanding and defending against poisoning of anomaly detectors. In *IMC* (2009), ACM, pp. 1–14.
- [46] SAAR-TSECHANSKY, M., AND PROVOST, F. Handling missing values when applying classification models. *JMLR* (2007).
- [47] SETTLES, B. Active learning literature survey. *University of Wisconsin, Madison* 52, 55–66 (1995), 11.
- [48] SHOKRI, R., AND SHMATIKOV, V. Privacy-preserving deep learning. In *CCS* (2015), ACM, pp. 1310–1321.
- [49] SMITH, T. W., MARSDEN, P., HOUT, M., AND KIM, J. General social surveys, 1972–2012, 2013.
- [50] STEVENS, D., AND LOWD, D. On the hardness of evading combinations of linear classifiers. In *AISec* (2013), ACM, pp. 77–86.
- [51] THEANO DEVELOPMENT TEAM. Theano: A Python framework for fast computation of mathematical expressions. *arXiv:1605.02688* (2016).
- [52] TOWELL, G. G., AND SHAVLIK, J. W. Extracting refined rules from knowledge-based neural networks. *Machine learning* 13, 1 (1993), 71–101.
- [53] VALIANT, L. G. A theory of the learnable. *Communications of the ACM* 27, 11 (1984), 1134–1142.
- [54] VINTERBO, S. Differentially private projected histograms: Construction and use for prediction. In *ECML-PKDD* (2012).
- [55] ŠRNĐIĆ, N., AND LASKOV, P. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP)* (2014), IEEE, pp. 197–211.
- [56] ZHANG, J., ZHANG, Z., XIAO, X., YANG, Y., AND WINSLETT, M. Functional mechanism: regression analysis under differential privacy. In *VLDB* (2012).
- [57] ZHU, J., AND HASTIE, T. Kernel logistic regression and the import vector machine. In *NIPS* (2001), pp. 1081–1088.

A Some Details on Models

SVMs. Support vector machines (SVMs) perform binary classification ($c = 2$) by defining a maximally separating hyperplane in d -dimensional feature space. A linear SVM is a function $f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + \beta)$ where ‘sign’ outputs 0 for all negative inputs and 1 otherwise. Linear SVMs are not suitable for non-linearly separable data. Here one uses instead kernel techniques [14].

A kernel is a function $K: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$. Typical kernels include the quadratic kernel $K_{\text{quad}}(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \cdot \mathbf{x}' + 1)^2$ and the Gaussian radial basis function (RBF) kernel $K_{\text{rbf}}(\mathbf{x}, \mathbf{x}') = e^{-\gamma \|\mathbf{x} - \mathbf{x}'\|^2}$, parameterized by a value $\gamma \in \mathbb{R}$. A kernel's projection function is a map ϕ defined by $K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}')$. We do not use ϕ explicitly, indeed for RBF kernels this produces an infinite-dimension vector. Instead, classification is defined using a “kernel trick”: $f(\mathbf{x}) = \text{sign}([\sum_{i=1}^t \alpha_i K(\mathbf{x}, \mathbf{x}_i)] + \beta)$ where β is again a learned threshold, $\alpha_1, \dots, \alpha_t$ are learned weights, and $\mathbf{x}_1, \dots, \mathbf{x}_t$ are feature vectors of inputs from a training set. The \mathbf{x}_i for which $\alpha_i \neq 0$ are called support vectors. Note that for non-zero α_i , it is the case that $\alpha_i < 0$ if the training-set label of \mathbf{x}_i was zero and $\alpha_i > 0$ otherwise.

Logistic regression. SVMs do not directly generalize to multiclass settings $c > 2$, nor do they output class probabilities. Logistic regression (LR) is a popular classifier that does. A binary LR model is defined as $f_1(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + \beta) = 1/(1 + e^{-(\mathbf{w} \cdot \mathbf{x} + \beta)})$ and $f_0(\mathbf{x}) = 1 - f_1(\mathbf{x})$. A class label is chosen as 1 iff $f_1(\mathbf{x}) > 0.5$.

When $c > 2$, one fixes c weight vectors $\mathbf{w}_0, \dots, \mathbf{w}_{c-1}$ each in \mathbb{R}^d , thresholds $\beta_0, \dots, \beta_{c-1}$ in \mathbb{R} and defines $f_i(\mathbf{x}) = e^{\mathbf{w}_i \cdot \mathbf{x} + \beta_i} / (\sum_{j=0}^{c-1} e^{\mathbf{w}_j \cdot \mathbf{x} + \beta_j})$ for $i \in \mathbb{Z}_c$. The class label is taken to be $\text{argmax}_i f_i(\mathbf{x})$. Multiclass regression is referred to as multinomial or softmax regression. An alternative approach to softmax regression is to build a binary model $\sigma(\mathbf{w}_i \cdot \mathbf{x} + \beta_i)$ per class in a *one-vs-rest* fashion and then set $f_i(\mathbf{x}) = \sigma(\mathbf{w}_i \cdot \mathbf{x} + \beta_i) / \sum_j \sigma(\mathbf{w}_j \cdot \mathbf{x} + \beta_j)$.

These are log-linear models, and may not be suitable for data that is not linearly separable in \mathcal{X} . Again, one may use kernel techniques to deal with more complex data relationships (c.f., [57]). Then, one replaces $\mathbf{w}_i \cdot \mathbf{x} + \beta_i$ with $\sum_{r=1}^t \alpha_{i,r} K(\mathbf{x}, \mathbf{x}_r) + \beta_i$. As written, this uses the entire set of training data points $\mathbf{x}_1, \dots, \mathbf{x}_t$ as so-called representors (here analogous to support vectors). Unlike with SVMs, where most training data set points will never end up as support vectors, here all training set points are potentially representors. In practice one uses a size $s < t$ random subset of training data [57].

Deep neural networks. A popular way of extending softmax regression to handle data that is non linearly separable in \mathcal{X} is to first apply one or more non-linear transformations to the input data. The goal of these *hidden layers* is to map the input data into a (typically) lower-dimensional space in which the classes are separable by the softmax layer. We focus here on fully connected networks, also known as multilayer perceptrons, with a single hidden layer. The hidden layer consists of a number h of *hidden nodes*, with associated weight vectors $\mathbf{w}_0^{(1)}, \dots, \mathbf{w}_{h-1}^{(1)}$ in \mathbb{R}^d and thresholds $\beta_0^{(1)}, \dots, \beta_{h-1}^{(1)}$ in \mathbb{R} . The i -th hidden unit applies a non linear transformation $h_i(\mathbf{x}) = g(\mathbf{w}_i^{(1)} \cdot \mathbf{x} + \beta_i^{(1)})$, where g is an activation function such as \tanh or σ . The vector $h(\mathbf{x}) \in \mathbb{R}^h$ is then

input into a softmax output layer with weight vectors $\mathbf{w}_0^{(2)}, \dots, \mathbf{w}_{c-1}^{(2)}$ in \mathbb{R}^h and thresholds $\beta_0^{(2)}, \dots, \beta_{c-1}^{(2)}$ in \mathbb{R} .

Decision trees. A decision tree T is a labeled tree. Each internal node v is labeled by a feature index $i \in \{1, \dots, d\}$ and a *splitting function* $\rho : \mathcal{X}_i \rightarrow \mathbb{Z}_{k_v}$, where $k_v \geq 2$ denotes the number of outgoing edges of v .

On an input $\mathbf{x} = (x_1, x_2, \dots, x_d)$, a tree T defines a computation as follows, starting at the root. When we reach a node v , labeled by $\{i, \rho\}$, we proceed to the child of v indexed by $\rho(x_i)$. We consider three types of splitting functions ρ that are typically used in practice ([11]):

- (1) The feature x_i is categorical with $\mathcal{X}_i = \mathbb{Z}_k$. Let $\{S, T\}$ be some partition of \mathbb{Z}_k . Then $k_v = 2$ and $\rho(x_i) = 0$ if $x_i \in S$ and $\rho(x_i) = 1$ if $x_i \in T$. This is a binary split on a categorical feature.
- (2) The feature x_i is categorical with $\mathcal{X}_i = \mathbb{Z}_k$. We have $k_v = k$ and $\rho(x_i) = x_i$. This corresponds to a k -ary split on a categorical feature of arity k .
- (3) The feature x_i is continuous with $\mathcal{X}_i = [a, b]$. Let $a < t < b$ be a *threshold*. Then $k_v = 2$ and $\rho(x_i) = 0$ if $x_i \leq t$ and $\rho(x_i) = 1$ if $x_i > t$. This is a binary split on a continuous feature with threshold t .

When we reach a leaf, we terminate and output that leaf’s value. This value can be a class label, or a class label and confidence score. This defines a function $f : \mathcal{X} \rightarrow \mathcal{Y}$.

B Details on Data Sets

Here we give some more information about the data sets we used in this work. Refer back to Table 3 and Table 5.

Synthetic data sets. We used 4 synthetic data sets from scikit [42]. The first two data sets are classic examples of non-linearly separable data, consisting of two concentric Circles, or two interleaving Moons. The next two synthetic data sets, Blobs and 5-Class, consist of Gaussian clusters of points assigned to either 3 or 5 classes.

Public data sets. We gathered a varied set of data sets representative of the type of data we would expect ML service users to use to train logistic and SVM based models. These include famous data sets used for supervised learning, obtained from the UCI ML repository (*Adult*, *Iris*, *Breast Cancer*, *Mushrooms*, *Diabetes*). We also consider the *Steak* and *GSS* data sets used in prior work on model inversion [23]. Finally, we add a data set of digits available in scikit, to visually illustrate training data leakage in kernelized logistic models (c.f. Section 4.1.3).

Public data sets and models from BigML. For experiments on decision trees, we chose a varied set of models publicly available on BigML’s platform. These models were trained by real MLaaS users and they cover a wide range of application scenarios, thus providing a realistic benchmark for the evaluation of our extraction attacks.

The *IRS* model predicts a US state, based on administrative tax records. The *Steak* and *GSS* models respectively predict a person’s preferred steak preparation and happiness level, from survey and demographic data. These two models were also considered in [23]. The *Email Importance* model predicts whether Gmail classifies an email as ‘important’ or not, given message metadata. The *Email Spam* model classifies emails as spam, given the presence of certain words in its content. The German Credit data set was taken from the UCI library [35] and classifies a user’s loan risk. Finally, two regression models respectively predict *Medical Charges* in the US based on state demographics, and the *Bitcoin Market Price* from daily opening and closing values.

C Analysis of the Path-Finding Algorithm

In this section, we analyze the correctness and complexity of the decision tree extraction algorithm in Algorithm 1. We assume that all leaves are assigned a unique id by the oracle \mathcal{O} , and that no continuous feature is split into intervals of width smaller than ε . We may use id to refer directly to the leaf with identity id .

Correctness. Termination of the algorithm follows immediately from the fact that new queries are only added to Q when a new leaf is visited. As the number of leaves in the tree is bounded, the algorithm must terminate.

We prove by contradiction that all leaves are eventually visited. Let the *depth* of a node v , denote the length of the path from v to the root (the root has depth 0). For two leaves id, id' , let A be their deepest common ancestor (A is the deepest node appearing on both the paths of id and id'). We denote the depth of A as $\Delta(\text{id}, \text{id}')$.

Suppose Algorithm 1 terminates without visiting all leaves, and let (id, id') be a pair of leaves with maximal $\Delta(\text{id}, \text{id}')$, such that id was visited but id' was not. Let x_i be the feature that their deepest common ancestor A splits on. When id is visited, the algorithm calls `LINE_SEARCH` or `CATEGORY_SPLIT` on feature x_i . As all leaf ids are unique and there are no intervals smaller than ε , we will discover a leaf in each sub-tree rooted at A , including the one that contains id' . Thus, we visit a leaf id'' for which $\Delta(\text{id}'', \text{id}') > \Delta(\text{id}, \text{id}')$, a contradiction.

Complexity. Let m denote the number of leaves in the tree. Each leaf is visited exactly once, and for each leaf we check all d features. Suppose continuous features have range $[0, b]$, and categorical features have arity k . For continuous features, finding one threshold takes at most $\log_2(\frac{b}{\varepsilon})$ queries. As the total number of splits on one feature is at most m (i.e., all nodes split on the same feature), finding all thresholds uses at most $m \cdot \log_2(\frac{b}{\varepsilon})$ queries. Testing a categorical feature uses k queries. The total query complexity is $O(m \cdot (d_{\text{cat}} \cdot k + d_{\text{cont}} \cdot m \cdot \log_2(\frac{b}{\varepsilon}))$, where d_{cat} and d_{cont} represent respectively the number of categorical and continuous features.

For the special case of boolean trees, the complexity is $O(m \cdot d)$. In comparison, the algorithm of [33], that uses membership queries only, has a complexity polynomial in d and 2^δ , where δ is the tree depth. For degenerate trees, 2^δ can be exponential in m , implying that the assumption of unique leaf identities (obtained from confidence scores for instance) provides an exponential speed-up over the best-known approach with class labels only. The algorithm from [33] can be extended to regression trees, with a complexity polynomial in the size of the output range \mathcal{Y} . Again, under the assumption of unique leaf identities (which could be obtained solely from the output values) we obtain a much more efficient algorithm, with a complexity independent of the output range.

The Top-Down Approach. The correctness and complexity of the top-down algorithm from Section 4.2 (which uses incomplete queries), follow from a similar analysis. The main difference is that we assume that all nodes have a unique id , rather than only the leaves.

D A Note on Improper Extraction

To extract a model f , without knowledge of the model class, a simple strategy is to extract a multilayer perceptron \hat{f} with a large enough hidden layer. Indeed, feed-forward networks with a single hidden layer can, in principle, closely approximate any continuous function over a bounded subset of \mathbb{R}^d [20, 28].

However, this strategy intuitively does not appear to be optimal. Even if we know that we can find a multilayer perceptron \hat{f} that closely matches f , \hat{f} might have a far more complex representation (more parameters) than f . Thus, tailoring the extraction to the ‘simpler’ model class of the target f appears more efficient. In learning theory, the problem of finding a succinct representation of some target model f is known as *Occam Learning* [13].

Our experiments indicate that such generic improper extraction indeed appears sub-optimal, in the context of equation-solving attacks. We train a softmax regression over the Adult data set with target “Race”. The model f is defined by 530 real-valued parameters. As shown in Section 4.1.2, using only 530 queries, we extract a model \hat{f} from the *same* model class, that closely matches f (\hat{f} and f predict the same labels on 100% of tested inputs, and produce class probabilities that differ by less than 10^{-7} in TV distance). We also extracted the same model, assuming a multilayer perceptron target class. Even with 1,000 hidden nodes (this model has 111,005 parameters), and 10 \times more queries (5,300), the extracted model \hat{f} is a weaker approximation of f (99.5% accuracy for class labels and TV distance of 10^{-2} for class probabilities).

Oblivious Multi-Party Machine Learning on Trusted Processors

Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta*, Sebastian Nowozin
Kapil Vaswani, Manuel Costa
Microsoft Research

Abstract

Privacy-preserving multi-party machine learning allows multiple organizations to perform collaborative data analytics while guaranteeing the privacy of their individual datasets. Using trusted SGX-processors for this task yields high performance, but requires a careful selection, adaptation, and implementation of machine-learning algorithms to provably prevent the exploitation of any side channels induced by data-dependent access patterns.

We propose data-oblivious machine learning algorithms for support vector machines, matrix factorization, neural networks, decision trees, and k-means clustering. We show that our efficient implementation based on Intel Skylake processors scales up to large, realistic datasets, with overheads several orders of magnitude lower than with previous approaches based on advanced cryptographic multi-party computation schemes.

1 Introduction

In many application domains, multiple parties would benefit from pooling their private datasets, training precise machine-learning models on the aggregate data, and sharing the benefits of using these models. For example, multiple hospitals might share patient data to train a model that helps in diagnosing a disease; having more data allows the machine learning algorithm to produce a better model, benefiting all the parties. As another example, multiple companies often collect complementary data about customers; sharing such data would allow machine learning algorithms to make joint predictions about customers based on a set of features that would not otherwise be available to any of the companies. This scenario also applies to individuals. For example, some systems learn an individual’s preferences to make accurate recommendations [9]; while users would like to keep their data private, they want to reap the rewards of correlating their preferences with those of other users.

Secure multi-party computation [17, 24, 44] and fully homomorphic encryption [23] are powerful cryptographic tools that can be used for privacy preserving machine learning. However, recent work [38, 47, 48, 54] reports large runtime overheads, which limits their prac-

tical adoption for compute-intensive analyses of large datasets. We propose an alternative privacy-preserving multi-party machine learning system based on trusted SGX processors [45]. In our system, multiple parties agree on a joint machine learning task to be executed on their aggregate data, and on an SGX-enabled data center to run the task. Although they do not trust one another, they can each review the corresponding machine-learning code, deploy the code into a processor-protected memory region (called an enclave), upload their encrypted data just for this task, perform remote attestation, securely upload their encryption keys into the enclave, run the machine learning code, and finally download the encrypted machine learning model. The model may also be kept within the enclave for secure evaluation by all the parties, subject to their agreed access control policies. Figure 1 provides an overview of our system.

While we rely on the processor to guarantee that only the machine learning code inside the enclave has direct access to the data, achieving privacy still requires a careful selection, adaptation, and implementation of machine-learning algorithms, in order to prevent the exploitation of any side channels induced by disk, network, and memory access patterns, which may otherwise leak a surprisingly large amount of data [39, 49, 70]. The robust property we want from these algorithms is *data-obliviousness*: the sequence of memory references, disk accesses, and network accesses that they perform should not depend on secret data. We propose data-oblivious machine learning algorithms for support vector machines (SVM), matrix factorization, neural networks, decision trees, and k-means clustering. Our data-oblivious algorithms are based on careful elimination of data-dependent accesses (SVM, neural networks and k-means), novel algorithmic techniques (matrix factorization) and deployment of platform specific hardware features (decision trees). We provide strong, provable confidentiality guarantees, similar to those achieved by purely cryptographic solutions: we ensure that an attacker that observes the sequence of I/O operations, including their addresses and their encrypted contents, cannot distinguish between two datasets of the same size that yield results of the same size.

We implemented and ran our algorithms on off-the-shelf Intel Skylake processors, using several large ma-

*Work done while at Microsoft Research; affiliated with Max Planck Institute for Software Systems (MPI-SWS), Germany.

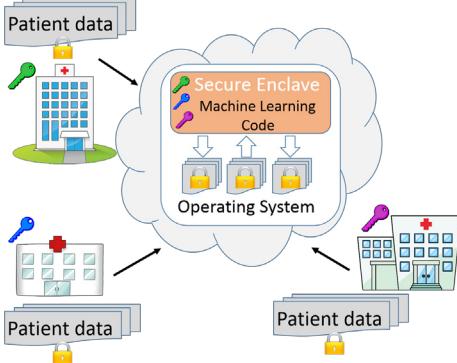


Figure 1: Sample privacy-preserving multi-party machine learning system. Multiple hospitals encrypt patient datasets, each with a different key. The hospitals deploy an agreed-upon machine learning algorithm in an enclave in a cloud data center and share their data keys with the enclave. The enclave processes the aggregate datasets and outputs an encrypted machine learning model.

chine learning datasets. Our results show that our approach scales to realistic datasets, with overheads that are several orders of magnitude better than with previous approaches based on advanced cryptographic multi-party computation schemes. On the other hand, our approach trusts the processor to protect the confidentiality of its internal state, whereas these cryptographic approaches do not rely on this assumption.

2 Preliminaries

Intel SGX SGX [45] is a set of new x86 instructions that applications can use to create protected memory regions within their address space. These regions, called enclaves, are isolated from any other code in the system, including operating system and hypervisor. The processor monitors all memory accesses to the enclaves: only code running in an enclave can access data in the enclave. When inside the physical processor package (in the processor’s caches), the enclave memory is available in plaintext, but it is encrypted and integrity protected when written to system memory (RAM). External code can only invoke code inside the enclave at statically-defined entry points. SGX also supports attestation and sealing [2]: code inside an enclave can get messages signed using a per-processor private key along with a digest of the enclave. This enables other entities to verify that these messages originated from a genuine enclave with a specific code and data configuration.

Using SGX instructions, applications can set up fine-grained trusted execution environments even in (potentially) hostile or compromised hosts, but application developers who write code that runs inside enclaves are still responsible for maintaining confidentiality of secrets managed by the enclave. In this paper, we focus on guaranteeing that the machine learning algorithms that we

load into enclaves do not leak information through memory, disk, or network access patterns.

Adversary Model We assume the machine learning computation runs in an SGX-enabled cloud data center that provides a convenient ‘neutral ground’ to run the computation on datasets provided by multiple parties. The parties do not trust one another, and they are also suspicious about the cloud provider. From the point of view of each party (or any subset of parties), the adversary models all the other parties and the cloud provider.

The adversary may control all the hardware in the cloud data center, except the processor chips used in the computation. In particular, the adversary controls the network cards, disks, and other chips in the motherboards. She may record, replay, and modify network packets or files. The adversary may also read or modify data after it left the processor chip using physical probing, direct memory access (DMA), or similar techniques.

The adversary may also control all the software in the data center, including the operating system and hypervisor. For instance, the adversary may change the page tables so that any enclave memory access results in a page fault. This active adversary is general enough to model privileged malware running in the operating or hypervisor layers, as well as malicious cloud administrators who may try to access the data by logging into hosts and inspecting disks and memory.

We assume that the adversary is unable to physically open and manipulate the SGX processor chips that run the machine learning computation. Denial-of-service and side-channel attacks based on power and timing analysis are outside our scope. We consider the implementation of the machine learning algorithms to be benign: the code will never intentionally try to leak secrets from enclaves. We assume that all parties agree on the machine learning code that gets access to their datasets, after inspecting the code or using automated verification [60] to ascertain its trustworthiness. We assume that all parties get access to the output of the machine learning algorithm, and focus on securing its implementation—limiting the amount of information released by its correct output [19] is outside the scope of this paper.

Security Guarantees We are interested in designing algorithms with strong provable security guarantees. The attacker described above should not gain any side information about sensitive data inputs. More precisely, for each machine learning algorithm, we specify *public parameters* that are allowed to be disclosed (such as the input sizes and the number of iterations to perform) and we treat all other inputs as private. We then say that an algorithm is *data-oblivious* if an attacker that interacts with it and observes its interaction with memory, disk and network learns nothing except possibly those public

parameters. We define this interaction as a trace execution τ of I/O events, each recording an access type (read or write), an address, and some contents, controlled by the adversary for all read accesses. Crucially, this trace leaks accurate information about access to code as well as data; for example, a conditional jump within an enclave may reveal the condition value by leaking the next code address [70].

We express our security properties using a simulation-based technique: for each run of an algorithm given some input that yields a trace τ , we show that there exists a *simulator program* given *only* the public parameters that simulates the interaction of the original algorithm with the memory by producing a trace τ' indistinguishable from τ . Intuitively, if the algorithm leaked any information depending on private data, then the simulator (that does not have the data) would not be able to adapt its behavior accordingly. Beside the public parameters, the simulator may be given the result of the algorithm (e.g., the machine learning model) in scenarios where the result is revealed to the parties running the algorithm. We rely on indistinguishability (rather than simple trace equivalence $\tau' = \tau$) to account for randomized algorithms, and in particular for encryption. For instance, any private contents in write events will be freshly encrypted and thus (under some suitable semantic-encryption security assumption) will appear to be independently random in both τ and τ' , rather than equal. More precisely, we define indistinguishability as usual in cryptography, using a game between a system that runs the algorithm (or the simulator) and a computationally bounded adversary that selects the inputs, interacts with the system, observes the trace, and attempts to guess whether it interacts with the algorithm or the simulator. The algorithm is data-oblivious when such adversaries guess correctly with probability at most $\frac{1}{2}$ plus a negligible advantage.

3 Data-Oblivious Primitives

Our algorithms rely on a library of general-purpose oblivious primitives. We describe them first and then show how we use them in machine learning algorithms.

Oblivious assignments and comparisons These primitives can be used to conditionally assign or compare integer, floating point, or 256-bit vector variables. They are implemented in x86-64 assembly, operating solely on registers whose content is loaded from and stored to memory using deterministic memory accesses. The registers are private to the processor; their contents are not accessible to code outside the enclave. As such, evaluations that involve registers only are not recorded in the trace τ , hence, any register-to-register data manipulation is data-oblivious by default.

We choose `omove()` and `ogreater()` as two representative oblivious primitives. In conjunction, they enable

Non-oblivious	Oblivious
<pre>int max(int x, int y) { if (x > y) return x; else return y; }</pre>	<pre>int max(int x, int y) { bool getx = ogreater(x, y); return omove(getx, x, y); }</pre>

Figure 2: **Left:** C++ function determining the maximum of two integers using a non-oblivious if-else statement; **right:** oblivious variant of the function using oblivious primitives.

the straightforward, oblivious implementation of the `max()` function, as shown in Figure 2. In the oblivious version of `max()`, `ogreater()` evaluates the guard $x > y$ and `omove()` selects either x or y , depending on that guard. In our library, similar to related work [53], both primitives are implemented with conditional instructions `cmovz` and `setg`. For example, in simplified form, `omove()` and `ogreater()` for 64-bit integers comprise the following instructions:

ogreater()	omove()
<pre>mov rcx, x mov rdx, y cmp rcx, rdx setg al retn</pre>	<pre>mov rcx, cond mov rdx, x mov rax, y test rcx, rcx cmovz rax, rdx retn</pre>

On top of such primitives for native C++ types, our library implements more complex primitives for user-defined types. For example, most of our oblivious algorithms rely on `omoveEx()`, an extended version of the basic `omove()`, which can be used to conditionally assign any type of variable; depending on the size of the given type, `omoveEx()` iteratively uses the 64-bit integer or 256-bit vector version of `omove()`.

Oblivious array accesses Scanning entire arrays is a commonly used technique to make data-dependent memory accesses oblivious. In the simplest case, we use `omoveEx()` iteratively to access each element when actually just a single element is to be loaded or stored.¹ However, our adversary model implies that, for enclave code, the attacker can only observe memory accesses at cache-line granularity. Accordingly, the x least significant bits² of memory addresses are not recorded in a trace τ . It is hence sufficient to scan arrays at cache-line granularity rather than element or byte granularity. We implement accordingly optimized array access primitives that leverage AVX2 vector instructions [31]. In particular, the `vpgatherdd` instruction can load each of the eight 32-bit (4-byte) components of a 256-bit vector

¹Dummy writes without actual effect are made to all but one element in case of a store. Modern processors treat such writes in the same way as real writes and mark corresponding cache lines as dirty.

²The value of x depends on the actual hardware implementation; for Skylake processors, where cache lines are 64 bytes long, $x = 6$.

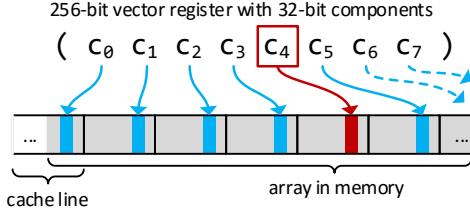


Figure 3: Optimized array scanning using the `vpgatherdd` instruction; here, the value of interest is read into C_4 . The other components perform dummy reads.

register from a different memory offset. Hence, by loading each component from a different cache line, 4 bytes can be read obliviously from an aligned 512-byte array with a single instruction as depicted in Figure 3 (i.e., a 4-byte read is hidden among 8 cache lines accessed via `vpgatherdd`). On top of this, the `oget()` primitive is created, which obliviously reads an element from an unaligned array of arbitrary form and size. `oget()` iteratively applies the `vpgatherdd` instruction in the described way while avoiding out-of-bounds reads. Depending on the dynamic layout of the caches, `oget()` can significantly speed-up oblivious array lookups (see Section 6.6). The construction of `oget()` is conservative in the sense that it assumes (i) that the processor may load vector components in arbitrary, possibly parallel order³ and (ii) that this order is recorded precisely in τ . For cases where (i) or (ii) do not apply, e.g., for software-only attackers, a further optimized version of `oget()` is described in Appendix B.

Oblivious sorting We implement oblivious sorting by passing its elements through a network of carefully arranged compare-and-swap functions. Given an input size n , the network layout is fixed and, hence, the memory accesses fed to the functions in each layer of the network depend only on n and not the content of the array (as opposed to, e.g., quicksort). Hence, its memory trace can be easily simulated using public parameter n and fixed element size. Though there exists an optimal sorting network due to Ajtai *et al.* [1], it incurs high constants. As a result, a Batcher’s sorting network [7] with running time of $O(n(\log n)^2)$ is preferred in practice. Our library includes a generic implementation of Batcher’s sort for shuffling the data as well as re-ordering input instances to allow for efficient (algorithm-specific) access later on. The sorting network takes as input an array of user-defined type and an oblivious compare-and-swap function for this type. The oblivious compare-and-swap usually relies on the `ogreater()` and `omoveEx()` primitives described above.

³The implementation of the `vpgatherdd` instruction is microarchitecture-specific and undocumented.

4 Machine Learning Algorithms

We describe five machine learning algorithms: four training and one prediction method, and their data-oblivious counter-parts. The algorithms vary in the complexity of access patterns, from randomly sampling the training data to input-dependent accesses to the corresponding model. Hence, we propose algorithm-specific mitigation techniques that build on the oblivious primitives from the last section.

4.1 K-Means

The goal of k-means clustering is to partition input data points into k clusters such that each point is assigned to the cluster closest to it. Data points are vectors in the Euclidean space \mathbb{R}^d . To implement clustering, we chose a popular and efficient Lloyd’s algorithm [40, 41, 43].

During its execution, k-means maintains a list of k points that represent the current cluster centroids: for $i = 1..k$, the i th point is the mean of all points currently assigned to the i th cluster. Starting from random centroids, the algorithm iteratively reassigns points between clusters: (1) for each point, it compares its distances to the current k centroids, and assigns it to the closest cluster; (2) once all points have been processed, it recomputes the centroids based on the new assignment. The algorithm ends after a fixed number of iterations, or once the clustering is stable, that is, in case points no longer change their cluster assignments. Depending on the application, k-means returns either the centroids or the assignment of data points to clusters.

Although the algorithm data flow is largely independent of the actual points and clusters, its naive implementation may still leak much information in the conditional update in (1)—enabling for instance an attacker to infer some point coordinates from the final assignment, or vice-versa—and in the recomputation (2)—leaking, for instance, intermediate cluster sizes and assignments.

In the following, we treat the number of points (n), clusters (k), dimension (d) and iterations (T) as public. We consider efficient, streaming implementations with, for each iteration, an outer loop traversing all points once, and successive inner loops on all centroids for the steps (1) and (2) above. For each centroid, in addition to the d coordinates, we locally maintain its current cluster size (in $0..n$). To perform both (1) and (2) in a single pass, we maintain both the current and the next centroids, and we delay the division of coordinates by the cluster size in the latter. Thus, for a given point, inner loop (1) for $i = 1..k$ maintains the (square of the) current minimal distance δ_{min} and its centroid index i_{min} . And inner loop (2) performs k conditional updates on the next centroids, depending on $i = i_{min}$. Finally, a single pass over centroids recomputes their coordinates. An important detail is to uniformly handle the special case of empty clus-

ters; another to select the initial centroids, for instance by sampling random points from the shuffled dataset.

In our adapted algorithm, the “privacy overhead” primarily consists of oblivious assignments to loop variables in (1), held in registers, and to the next centroids, held in the cache. In particular, instead of updating statistics for only the centroid that the point belongs to, we make dummy updates to each centroid (using our `omoveEx()` primitive). In the computation of new centroids, we use a combination of `ogreater()` and `omoveEx()` to handle the case of empty clusters. These changes do not affect the algorithm’s time complexity: in the RAM model the operations above can still be done in $O(T(nkd + kd)) = O(Tnkd)$ operations.

Theorem 1. *The adapted k-means algorithm runs in time $O(Tnkd)$ and is data-oblivious, as there exists a simulator for k-means that depends only on T , n , d , and k .*

Proof. The simulator can be trivially constructed as follows: given T , n , d and k , it chooses n random points from \mathbb{R}^d and simply runs the algorithm above for k centroids and T iterations. \square

It is easy to see that the subroutine for finding the closest centroid in the training algorithm can be also used to predict the trained cluster that an input point belongs to.

4.2 Supervised Learning Methods

In supervised machine learning problems, we are given a dataset $D = \{(x_i, y_i)\}_{i=1..n}$ of instances, where $x_i \in \mathcal{X}$ is an observation and $y_i \in \mathcal{Y}$ is a desired prediction. The goal then is to learn a predictive model $f: \mathcal{X} \rightarrow \mathcal{Y}$ such that $f(x_i) \approx y_i$ and the model generalizes to unseen instances $x \in \mathcal{X}$. Many machine learning methods learn such a model by minimizing an *empirical risk* objective function together with a regularization term [65]:

$$\min_w \Omega(w) + \frac{1}{n} \sum_{i=1}^n L(y_i, f_w(x_i)). \quad (1)$$

We will show secure implementations of support vector machines (SVM) and neural networks, which are of the form (1). Other popular methods such as linear regression and logistic regression are also instances of (1).

Most algorithms to minimize (1) operate iteratively on small subsets of the data at a time. When sampling these subsets, one common requirement for correctness is that the algorithm should have access to a distribution of samples with an unbiased estimate of the expected value of the original distribution. We make an important observation that an unbiased estimate of the expected value of a population can be computed from a subset of independent and identically distributed instances as well as from a subset of *pairwise-distinct* instances.

Thus, we can achieve correctness and security by adapting the learning algorithm as follows. Repeatedly, (1) securely shuffle all instances at random, using Batcher’s sort, for example, or an oblivious shuffle [50]; (2) run the learning algorithm on the instances *sequentially*, rather than randomly, either individually or in small batches. Thus, the cost of shuffling is amortized over all n instances. Next, we illustrate this scheme for support vector machines and neural networks.

4.3 Support Vector Machines (SVM)

Support Vector Machines are a popular machine learning model for classification problems. The original formulation [12] applies to problems with two classes. Formally, SVM specializes (1) by using the linear model $f_w(x) = \langle w, x \rangle$, the regularization $\Omega(w) = \frac{\lambda}{2} \|w\|^2$ for $\lambda > 0$, and the loss function $L(y_i, f_w(x_i)) = \max\{0, 1 - y_i f_w(x_i)\}$.

The SVM method is important historically and in practice for at least four separate reasons: *first*, it is easy to use and tune and it performs well in practice; *second*, it is derived from the principle of structural risk minimization [65] and comes with excellent theoretical guarantees in the form of generalization bounds; *third*, it was the first method to be turned into a non-linear classifier through application of the kernel trick [12, 56], *fourth*, the SVM has inspired a large number of generalizations, for example to the multi-class case [67], regression [61], and general pattern recognition problems [63].

Here we only consider the linear (primal) case with two classes, but our methods would readily extend to multiple classes or support vector regression problems. There are many methods to solve the SVM objective and for its simplicity we adapt the state-of-the-art *Pegasos* method [58]. The algorithm proceeds in iterations $t = 1..T$ and, at each iteration, works on small subsets of l training instances at a time, $A^{(t)}$. It updates a sequence of weight vectors $w^{(1)}, w^{(2)}, \dots, w^{(T)}$ converging to the optimal minimizer of the objective function (1).

Let us now consider in detail our implementation of the algorithm, and the changes that make it data-oblivious. We present the pseudo-code in Algorithm 1 where our changes are highlighted in blue, and indented to the right. As explained in Section 4.2, SVM samples input data during training. Instead, we obliviously (or privately) shuffle the data and process it sequentially. The original algorithm updates the model using instances that are mispredicted by the current model, $A_+^{(t)}$ in Line 5 of the pseudo-code. As this would reveal the state of the current model to the attacker, we make sure that the computation depends on every instance of $A^{(t)}$. In particular, we generate a modified set of instances in $B^{(t)}$ which has the original (x, y) instance if x is mispredicted and $(x, 0)$ otherwise, assigning either 0 or y using our `ogreater()` and `omove()` primitives (see Figure 2).

Algorithm 1 SVM Original with changes (starting with \triangleright) and additional steps required for the Oblivious Version indicated in blue.

```

1: INPUT:  $I = \{(x_i, y_i)\}_{i=1,\dots,n}$ ,  $\lambda, T, l$ 
2: INITIALIZE: Choose  $w^{(0)}$  s.t.  $\|w^{(0)}\| \leq 1/\sqrt{\lambda}$ 
3:   Shuffle  $I$ 
4: FOR  $t = 1, 2, \dots, T \times n/l$ 
5:   Choose  $A^{(t)} \subseteq I$  s.t.  $|A^{(t)}| = l$ 
      $\triangleright$  Set  $A^{(t)}$  to  $t$ th batch of  $l$  instances
6:   Set  $A_+^{(t)} = \{(x, y) \in A^{(t)} : y\langle w^t, x \rangle < 1\}$ 
      $\triangleright B^{(t)} = \{(x, \mathbf{1}[y\langle w^t, x \rangle < 1]y) : \forall (x, y) \in A^{(t)}\}$ 
7:   Set  $\eta = 1/\lambda t$ 
8:   Set  $v = \sum_{(x,y) \in A_+^{(t)}} yx$             $\triangleright v = \sum_{(x,z) \in B^{(t)}} zx$ 
9:   Set  $v = (1 - \eta \lambda)w^{(t)} + \frac{\eta}{l}v$ 
10:  Set  $c = 1 < \frac{1}{\sqrt{\lambda}}\|w\|$ 
11:  Set  $w^{(t+1)} = \min\left\{1, \frac{1}{\sqrt{\lambda}}\|w\|\right\}v$ 
      $\triangleright$  Set  $w^{(t+1)} = \left(c + (1 - c) \times \frac{1}{\sqrt{\lambda}\|w\|}\right)v$ 
12: OUTPUT  $w^{(t+1)}$ 

```

The second change is due to a Euclidean projection in Line 11, where v is multiplied by the minimum of the two values. In the oblivious version, we ensure that both values participate in the update of the model, again using our oblivious primitives. The modifications above are simple and, if the data is shuffled offline, asymptotically do not add overhead as the algorithm has to perform prediction for every value in the sample. Otherwise, the overhead of sorting is amortized as T is usually set to at least one.

Theorem 2. *The SVM algorithm described above runs in time $O(n(\log n)^2)$ and is data-oblivious, as there exists a simulator for SVM that depends only on T, n, d, λ and l , where d is the number of features in each input instance.*

The simulator can be constructed by composing a simulator for oblivious sorting and one that follows the steps of Algorithm 1.

We note that the oblivious computation of a label in the training algorithm ($\langle w, x \rangle$ in Line 6 in Algorithm 1) can be used also for the prediction phase of SVM.

4.4 Neural Networks

Feedforward neural networks are classic models for pattern recognition that process an observation using a sequence of learned non-linear transformations [10]. Recently, deep neural networks made significant progress on difficult pattern recognition applications in speech, vision, and natural language understanding and the collective set of methods and models is known as *deep learning* [26].

Formally, a feedforward neural network is a sequence of transformations $f(x) = f_t(\dots f_2(f_1(x)))$, where each transformation f_i is described by a fixed family of transformations and a parameter w_i to identify one particular element in that family. *Learning* a neural network means to find suitable parameters by minimization of the learning objective (1).

To minimize (1) efficiently in the context of neural networks, we use stochastic gradient methods (SGD) on small subsets of training data [26]. In particular, for $l \ll n$, say $l = 32$, we compute a *parameter gradient* on a subset $S \subset \{1, 2, \dots, n\}$, $|S| = l$ of the data as

$$\nabla_w \Omega(w) + \frac{1}{l} \sum_{i \in S} \nabla_w L(y_i, f(x_i)). \quad (2)$$

The expression above is an unbiased estimate of the gradient of (1) and we can use it to update the parameters via gradient descent. By repeating this update for many subsets S we can find parameters that approximately minimize the objective. Instead, as for SVM, we use disjoint subsets that are contiguous within the set of all (obliviously or privately) shuffled instances and iterate T times.

Because most neural networks densely process each input instance, memory access patterns during training and testing do not depend on the particular data instance. There are two exceptions. First, the initialization of a vector with $|\mathcal{Y}|$ ground truth labels depends on the true label y_i of the instance (recall that \mathcal{Y} is the set of possible prediction classes or labels). In particular, the y_i th entry is set, for example, to 1 and all other entries to 0. We initialize the label vector and hide the true label of the instance by using our oblivious comparison and move operations. The second exception is due to special functions that occur in certain f_i , for example in tanh-activation layers. Since special functions are relatively expensive, they are usually evaluated using piecewise approximations. Such conditional computation may leak parameter values and, in our adapted algorithm, we instead compute the approximation obliviously using a sequence of oblivious move operations. Neither of these changes affects the complexity of the algorithm.

The prediction counterpart of NN, similar to the k-means and SVM algorithms, is a subroutine of the training algorithm. Hence, our changes can be also used to make an oblivious prediction given a trained network.

4.5 Decision Tree Evaluation

Decision trees are common machine learning models for classification and regression tasks [15, 51, 52]. In these models, a tree is traversed from root to leaf node by performing a simple test on a given instance, at each interior node of the tree. Once a leaf node is reached, a simple model stored at this node, for example a constant value, is used as prediction.

Decision trees remain popular because they are *non-parametric*: the size of the decision tree can grow with more training data, providing increasingly accurate models. Ensembles of decision trees, for example in the form of *random forests* [14] offer improved predictive performance by averaging the predictions of many individual tree models [16]. Decision trees illustrate a class of data structures whose usage is highly instance-specific: when evaluating the model, the path traversed from root to leaf node reveals a large amount of information on both the instance and the tree itself. To enable the evaluation of decision trees without leaking side information, we adapt the evaluation algorithm of an existing library for random forests to make it data oblivious. We keep modifications of the existing implementation at a minimum, relying on the primitives of Section 3 wherever possible. Our target tree evaluation algorithm operates on one instance $x \in \mathbb{R}^d$ at a time. In particular, the trees are such that, at each interior node, a simple *decision stump* is performed:

$$\phi(x; j, t) = \begin{cases} \text{left}, & \text{if } x(j) \leq t, \\ \text{right}, & \text{otherwise,} \end{cases} \quad (3)$$

where $j \in \{1, \dots, d\}$ and $t \in \mathbb{R}$ are learned parameters stored at the interior tree node.

In order to conceal the path taken through a tree, we modify the algorithm such that each tree layer is stored as an array of nodes. During evaluation of an instance x , the tree is traversed by making exactly one oblivious lookup in each of these arrays. At each node, the lookup $x(j)$ and corresponding floating point comparison are done using our oblivious primitives. In case a leaf is found early, that is before the last layer of the tree was reached, its ID is stored obliviously and the algorithm proceeds with dummy accesses for the remaining layers of the tree. The predictions of all trees in a random forest are accumulated obliviously in an array; the final output is the prediction with the largest weight.

Together, the described modifications guarantee data-obliviousness both for instances and for trees (of the same size, up to padding). The algorithmic overhead is linear in the number of nodes n in a tree, i.e., $O(n)$ for a fixed d ; we omit the corresponding formal development.

4.6 Matrix Factorization

Matrix factorization methods [55] are a popular set of techniques for constructing recommender systems [32]. Given *users* and *items* to be rated, we take as input the observed ratings for a fraction of user-item pairs, either as explicit scores (“five stars”) or implicit user feedback. As a running example, we consider a system to recommend movies to viewers based on their experience.

Matrix factorization embeds users and items into a latent vector space, such that the inner product of a user vector with an item vector produces an estimate of the

rating a user would assign to the item. We can then use this expected rating to propose novel items to the user. While the individual preference dimensions in the user and item vectors are not assigned fixed meanings, empirically they often correspond to interpretable properties of the items. For example, a latent dimension may correspond to the level of action the movie contains.

Matrix factorization methods are remarkably effective [9] because they learn to transfer preference information across users and items by discovering dimensions of preferences shared by all users and items.

Let n be the number of users and m the number of items in the system. We may represent all (known and unknown) ratings as a matrix $R \in \mathbb{R}^{n \times m}$. The input consists of M ratings $r_{i,j}$ with $i \in 1..n$ and $j \in 1..m$, given by users to the items they have seen (using the movies analogy). The output consists of $U \in \mathbb{R}^{n \times d}$ and $V \in \mathbb{R}^{m \times d}$ such that $R \approx UV^\top$; these two matrices may then be used to predict unknown ratings $r_{i,j}$ as inner products $\langle u_i, v_j \rangle$. Following [48], we refer to u_i and v_j as user and item profiles, respectively.

The computation of U and V is performed by minimizing regularized least squares on the known ratings:

$$\min \frac{1}{M} \sum (r_{i,j} - \langle u_i, v_j \rangle)^2 + \lambda \sum \|u_i\|_2^2 + \mu \sum \|v_j\|_2^2 \quad (4)$$

where λ and μ determine the extent of regularization. The function above is not jointly convex in U and V , but becomes strictly convex in U for a fixed V , and strictly convex in V for a fixed U .

We implement matrix factorization using a gradient descent, as in prior work on oblivious methods [47, 48]. More efficient methods are now available to solve (4), such as the so-called *damped Wiberg method*, as shown in an extensive empirical evaluation [30], but they all involve more advanced linear algebra, so we leave their privacy-preserving implementation for future work.

Gradient descent This method iteratively updates U and V based on the current prediction error on the input ratings. The error is computed as $e_{i,j} = r_{i,j} - \langle u_i, v_j \rangle$, and u_i and v_j are updated in the opposite direction of the gradient as follows:

$$u_i^{(t+1)} \leftarrow u_i^{(t)} + \gamma \left[\sum_j e_{i,j} v_j^{(t)} - \lambda u_i^{(t)} \right] \quad (5)$$

$$v_j^{(t+1)} \leftarrow v_j^{(t)} + \gamma \left[\sum_i e_{i,j} u_i^{(t)} - \mu v_j^{(t)} \right] \quad (6)$$

The descent continues as long as the error (4) decreases, or for a fixed number of iterations (T). Each iteration can be efficiently computed by updating U and V sequentially. To update each user profile u_i , we may for instance use an auxiliary linked list of user ratings and pointers to the corresponding movie profiles in V .

The gradient descent above runs in time $\Theta(TM)$ in the RAM model, since all ratings are used at each iteration. For a fixed number of iterations, the access pattern of the algorithm does not depend on the actual values of the input ratings. However, it still reveals much sensitive information about which user-item pairs appear in the input ratings. For example, assuming they indicate which users have seen which movies, it trivially reveals the popularity of each movie, and the intersection of movie profiles between users, during the gradient update (see [46] for privacy implications of leaking movie ratings).

Our data-oblivious algorithm We design an algorithm whose observable behaviour depends only on public parameters n, m, M and T , and, hence, it can be simulated and does not reveal R . (We assume that d, λ, μ , and γ are public and do not depend on the input data.)

The high level idea is to use data structures that interleave user and movie profiles. This interleaving allows us to perform an update by sequentially reading and updating these profiles in-place. Once all profiles have been updated, some additional processing is required to interleave them for the next iteration but, with some care, this can also be implemented by sequential traversals of our data structures. (An illustration of the algorithm can be found in the Appendix.)

Our algorithm preserves the symmetry between users and items. It maintains data structures \mathbf{U} and \mathbf{V} that correspond to expanded versions of the matrices U and V . Intuitively, every user profile in \mathbf{U} is followed by the movie profiles required to update it (that is, the profiles for all movies rated by this user), and symmetrically every movie profile in \mathbf{V} is followed by its user profiles. We use superscript notation $\mathbf{U}^{(t)}$ and $\mathbf{V}^{(t)}$ to distinguish these data structures between iterations.

\mathbf{U} stores n *user tuples* that embed the user profiles of the original U , and M *rating tuples* that contain both movie profiles and their ratings. All tuples have the same size; they each include a *user id*, a *movie id*, a *rating*, and a vector of d values. User tuples are of the form $(i, 0, 0, u_i)$ with $i \in 1..n$; Rating tuples are of the form $(i, j, r_{i,j}, v_j)$. Hence, for each rating for j , we have a copy of v_j in a rating tuple. \mathbf{V} symmetrically stores m *item tuples*, of the form $(0, j, 0, v_j)$, and M *rating tuples*, of the form $(i, j, r_{i,j}, u_i)$.

The precise ordering of tuples within \mathbf{U} (and \mathbf{V}) is explained shortly but, as long as the tuples of \mathbf{U} are grouped by user ids (i), and the user tuple precedes its rating tuples, we can compute each $u_i^{(t+1)}$ according to Equation (5) by traversing $\mathbf{U}^{(t)}$ once, in order. After an initial *Setup*, each iteration actually consists of three data-oblivious phases:

- Update the user profiles u_i within $\mathbf{U}^{(t)}$ using Equation (5); let $\tilde{\mathbf{U}}$ be the updated data structure;

- Extract $U^{(t+1)}$ from $\tilde{\mathbf{U}}$;
- Copy $U^{(t+1)}$ into the rating tuples of $\tilde{\mathbf{V}}$ to obtain $\mathbf{V}^{(t+1)}$ for the next iteration.

(We omit symmetric steps producing $\tilde{\mathbf{V}}$, $\mathbf{V}^{(t+1)}$, and $\mathbf{U}^{(t+1)}$.) The extraction step is necessary to compute the prediction error and prepare \mathbf{U} and \mathbf{V} for the next iteration. Without tweaking the tuple ordering, the only efficient way of doing so would be to sort $\tilde{\mathbf{U}}$ lexicographically (by j , then i) so that the updated user profiles appear in the first n tuples (the approach taken in [48]). Oblivious sorting at each iteration is expensive, however, and would take $O((M+n)(\log(M+n))^2)$ oblivious compare-and-swap of pairs of $d+3$ elements.

Instead, we carefully place the user tuples in $\tilde{\mathbf{U}}$ so that they can be extracted in a single scan, outputting profiles *at a fixed rate*: one user profile every $(M+n)/n$ tuples, on average. Intuitively, this is achieved by interleaving the tuples of users with many ratings with those of users with few ratings—Section 4.7 explains how we efficiently compute such a tuple ordering.

Setup phase: We first initialize the user and vector profiles, and fill \mathbf{U} and \mathbf{V} using the input ratings.

(1) We build a sequence $L_{\mathbf{U}}$ (and symmetrically $L_{\mathbf{V}}$) that, for every user, contains a pair of the user id i and the count w_i of the movies he has rated. To this end, we extract the user ids from the input ratings (discarding the other fields); we sort them; we rewrite them sequentially, so that each entry is extended with a partial count and a flag indicating whether the next user id changes; and we sort them again, this time by flag then user id, to obtain $L_{\mathbf{U}}$ as the top n entries. (Directly outputting $L_{\mathbf{U}}$ during the sequential scan would reveal the counts.) For instance, after the first sorting and rewriting, the entries may be of the form $(1, 1, \perp), (1, 2, \perp), (1, 3, \top), (2, 1, \perp), \dots$

(2) We expand $L_{\mathbf{U}}$ (and symmetrically $L_{\mathbf{V}}$) into a sequence $I_{\mathbf{U}}$ of size $M+n$ that includes, for every user i with w_i ratings, one tuple $(i, 0, \perp, k, \ell)$ for each $k = 0..w_i$, such that the values ℓ are ordered by the interleaving explained in Section 4.7.

(3) We construct \mathbf{U} with empty user and rating profiles, as follows. Our goal is to order the input ratings according to $L_{\mathbf{U}}$. To this end, we extend each input rating with a user-rating sequence number $k = 1..w_i$, thereby producing M tuples $(i, j, r_{i,j}, k, \perp)$, and we append those to $I_{\mathbf{U}}$. We sort those tuples by i then k then $r_{i,j}$, so that $(i, 0, \perp, k, \ell)$ is directly followed by $(i, j, r_{i,j}, k, \perp)$ for $k = 1..w_i$; we sequentially rewrite those tuples so that they become $(_, _, _, _, _)$ directly followed by $(i, j, r_{i,j}, k, \ell)$; we sort again by ℓ ; and we discard the last M dummy tuples $(_, _, _, _, _)$.

(4) We generate initial values for the user and item profiles by scanning \mathbf{U} and filling in u_i and v_j using two pseudo-random functions (PRFs): one for u_i s and one

for v_j s. For each, user tuple $(i, 0, 0, u_i)$, we use the first PRF on inputs $(i-1)d+1, \dots, id$ to generate d random numbers that we normalize and write to u_i . For each, rating tuple $(i, j, r_{i,j}, v_j)$, we use the second PRF on inputs $(j-1)d+1, \dots, jd$ to generate d random numbers that we also normalize and write to v_j . We then use the same two PRFs for \mathbf{V} : the first one for rating tuples and the second one for item tuples.

Update phase: We compute updated user profiles (and symmetrically item profiles) in a single scan, reading each tuple of \mathbf{U} (and symmetrically \mathbf{V}) and (always) rewriting its vector—that is, its last d values, storing u_i for user tuples and v_j for rating tuples.

We use 4 loop variables u , δ , u° , and δ° each holding a \mathbb{R}^d vector, to record partially-updated profiles for the current user and for user i° . We first explain how u and δ are updated for the current user (i). During the scan, upon reading a user tuple $(i, 0, 0, u_i)$, as is always the case for the first tuple, we set u to u_i and δ to $u_i(1 - \lambda\gamma)$ and we overwrite u_i (to hide the fact that we scanned a user tuple). Upon reading a rating tuple $(i, j, r_{i,j}, v_j)$ for the current user i , we update δ to $\gamma v_j(r_{i,j} - \langle u, v_j \rangle) + \delta$ and overwrite v_j with δ . Hence, the last rating tuple (before the next user tuple) now stores the updated profile $u_i^{(t+1)}$ for the current user i .

We now bring our attention to i° , u° , and δ° . Recall that our interleaving of users in \mathbf{U} splits the rating tuples for some users. In such cases, if there are ratings left to scan, the running value δ written to v_j (before scanning the next user) may not yet contain the updated user profile. Accordingly, we use i° , u° , and δ° to save the state of the ‘split’ user while we process the next user, and restore it later as we scan the next rating tuple of the form $(i^\circ, j, r_{i^\circ,j}, v_j)$. In the full version of the paper we prove that a single state copy suffices during the expansion of $L_{\mathbf{U}}$ as we split at most one user at a time.

Extraction phase: The update leaves some of the values $u_i^{(t+1)}$ scattered within $\tilde{\mathbf{U}}$ (and similarly for $v_j^{(t+1)}$ within $\tilde{\mathbf{V}}$). Similar to the update phase we can extract all profiles by maintaining a state i° and u° for only one user. We extract U while scanning $\tilde{\mathbf{U}}$. In particular, after reading the last tuple of every chunk of size $(M+n)/n$ in $\tilde{\mathbf{U}}$ we always append an entry to U . This entry is either i° and u° or the content of the last tuple i and u . Meanwhile, after reading every tuple of $\tilde{\mathbf{U}}$ we write back either the same entry or the profile that was written to U last. This step ensures that user tuples contain the updated $u^{(t+1)}$. We also update i° and u° on every tuple: either performing a dummy update or changing the state to the next (split) user.

This step relies on a preliminary re-ordering and interleaving of users, such that the i th chunk of tuples always contains (a copy of) a user profile, and all n user profiles

can be collected (details of the expansion properties that are used here are described in the following section and in the full version of the paper).

Copying phase: We finally propagate the updated user profiles $U^{(t+1)}$ to the rating tuples in $\tilde{\mathbf{V}}$, which still carry (multiple copies of) the user profiles $U^{(t)}$. We update $\tilde{\mathbf{V}}$ sequentially in chunks of size n , that is, we first update the first n rows of \mathbf{V} , then rows $n+1$ to $2n$ and so on until all \mathbf{V} is updated, each time copying from the same n user profiles of $U^{(t+1)}$, as follows. (The exact chunk size is irrelevant, but n is asymptotically optimal.)

Recall that each rating tuple of $\tilde{\mathbf{V}}$ is of the form $(i, j, r_{i,j}, u_i^\ell, \ell)$ where $i \neq 0$ and ℓ indicates the interleaved position of the tuple in \mathbf{V} . To each chunk of $\tilde{\mathbf{V}}$, we append the profiles of $U^{(t+1)}$ extended with dummy values, of the form $(i, 0, -, u_i^{(t+1)}, -)$; we sort those $2n$ tuples by i then j , so that each tuple from $U^{(t+1)}$ immediately precedes tuples from (the chunk of) $\tilde{\mathbf{V}}$ whose user profile must be updated by $u_i^{(t+1)}$; we perform all updates by a linear rewriting; we sort again by ℓ ; and we keep the first n tuples. Finally, $\mathbf{V}^{(t+1)}$ is just the concatenation of those updated chunks.

Theorem 3. *Our matrix factorization algorithm runs in time $O((M+\tilde{n})(\log(M+\tilde{n}))^2 + T(M+\tilde{n})(\log \tilde{n})^2)$ where $\tilde{n} = \max(n,m)$. It is data-oblivious, as there exists a simulator that depends only on T, M, n, m , and d and produces the same trace.*

Proof Outline. The Setup phase is the most expensive, as it involves oblivious sorting on all the input ratings at once, with a $O((M+\tilde{n})(\log(M+\tilde{n}))^2)$ run time. The update phase runs in time $O(M+n+m)$ since it requires a single scan of \mathbf{U} and \mathbf{V} . The extraction phase similarly runs in time $O(M+n+m)$. The copying phase runs in time $O((M+m)(\log n)^2 + (M+n)(\log m)^2)$ due to $(M+m)/n$ sorts of U of size n and $(M+n)/m$ sorts of V of size m . Since all phases except Setup run T times, the total run time is

$$O((M+\tilde{n})\log^2(M+\tilde{n}) + T(M+\tilde{n})\log^2\tilde{n}).$$

A simulator can be built from the public parameters mentioned in the beginning of the algorithm. It executes every step of the algorithm: it creates the interleaving data structures that depend only on n, m, M and d and updates them using the steps of the Setup once and runs the Update, Extraction and Copy phases for T iterations. As part of Setup, it invokes the simulator of the sequence expansion algorithm described in the full version of the paper. \square

4.7 Equally-Interleaved Expansion

We finally present our method for arranging tuples of \mathbf{U} and \mathbf{V} in Matrix Factorization. We believe this method is

applicable to other data processing scenarios. For example, Arasu and Kaushik [4] use a similar, careful arrangement of tuples to obliviously answer database queries.

Definition 1. A weighted list L is a sequence of pairs (i, w_i) with n elements i and integer weights $w_i \geq 1$.

An expansion I of L is a sequence of elements of length $\sum^n w_i$ such that every element i occurs exactly w_i times.

Definition 2. Let $\alpha = \sum w_i / n$ be the average weight of L and the j th chunk of I be the sub-sequence of $\lceil \alpha \rceil$ elements $I_{\lfloor (j-1)\alpha + 1 \rfloor}, \dots, I_{\lceil j\alpha \rceil}$.

I equally interleaves L when all its elements can be collected by selecting one element from each chunk.

For example, for $L = (a, 4), (b, 1), (c, 1)$, every chunk has $\alpha = 2$ elements. The expansion $I = a, b, a, c, a, a$ equally interleaves L , as its elements a , b , and c can be chosen from its third, first, and second chunks, respectively. The expansion $I' = a, a, a, a, b, c$ does not.

We propose an efficient method for generating equal interleavings. Since it is used as an oblivious building block, we ensure that it accesses L , I and intermediate data structures in a manner that depends only on n and $M = \sum w_i$, not on the individual weights. (In matrix factorization, M is the total number of input ratings.) We adopt the terminology of Arasu and Kaushik [4], even if our definitions and algorithm are different. (In comparison, our expansions do not involve padding, as we do not require that copies of the same element are adjacent).

Given a weighted list L , we say that element i is heavy when $w_i \geq \alpha$, and light otherwise. The main idea is to put at most one light element in every chunk, filling the rest with heavy elements. We proceed in two steps: (1) we reorder L so that each heavy element is followed by light elements that compensate for it; (2) we sequentially produce chunks containing copies of one or two elements.

Step 1: Assume L is sorted by decreasing weights ($w_i \geq w_{i+1}$ for $i \in [1, n - 1]$), and b is its last heavy element ($w_b \geq \alpha > w_{b+1}$). Let δ_i be the sum of differences defined as $\sum_{j \in [1, i]} (w_j - \alpha)$ for heavy elements and $\sum_{j \in [b+1, i]} (\alpha - w_j)$ for light elements. Let S be L (obliviously) sorted by δ_j , breaking ties in favor of higher element indices. This does not yet guarantee that light elements appear after the heavy element they compensate for. To this end, we scan S starting from its last element (which is always the lightest), swapping any light element followed by a heavy element (so that, eventually, the first element is the heaviest).

Step 2: We produce I sequentially, using two loop variables: k , the latest heavy element read so far; and w , the remaining number of copies of k to place in I . We repeatedly read an element from the re-ordered list and produce a chunk of elements. For the first element k_1 , we produce α copies of k_1 , and we set $k = k_1$ and

$w = w_{k_1} - \alpha$. For each light element k_i , we produce w_{k_i} copies of k_i and $\alpha - w_{k_i}$ copies of k , and we decrement w by $\alpha - w_{k_i}$. For each heavy element k_i , we produce w copies of k and $\alpha - w$ copies of k_i , and we set $k = k_i$ and $w = w_{k_i} - (\alpha - w)$.

Continuing with our example sequence L above, a is heavy, b and c are light, and we have $\delta_a = 2$, $\delta_b = 1$, and $\delta_c = 2$. Sorting L by δ yields $(b, 1), (a, 4), (c, 1)$. Swapping heavy and light elements yields $(a, 4), (b, 1), (c, 1)$ and we produce the expansion $I = a, a, b, a, c, a$.

In the full version of the paper we prove that the algorithm is oblivious, always succeeds and runs in time $O(n(\log n)^2 + \sum w)$.

5 Protocols

For completeness, we give an overview of the protocols we use for running multi-party machine learning algorithms in a cloud equipped with SGX processors. Our protocols are standard, and similar to those used in prior work for outsourcing computations [29, 57]. For simplicity, we describe protocols involving a single enclave.

We assume that each party agrees on the machine-learning code, its public parameters, and the identities of all other parties (based, for example, on their public keys for signature). One of the parties sends this collection of code and static data to the cloud data center, where an (untrusted) code-loader allocates resources and creates an enclave with that code and data.

Each party independently establishes a secure channel with the enclave, authenticating themselves (e.g., using signatures) and using remote attestation [2] to check the integrity of the code and static data loaded into the enclave. They may independently interact with the cloud provider to confirm that this SGX processor is part of that data center. Each party securely uploads its private data to the enclave, using for instance AES-GCM for confidentiality and integrity protection. Each party uses a separate, locally-generated secret key to encrypt its own input data set, and uses its secure channel to share that key with the enclave. The agreed-upon machine learning code may also be optionally encrypted but we expect that in the common case this code will be public.

After communicating with all parties, and getting the keys for all the data sets, the enclave code runs the target algorithm on the whole data set, and outputs a machine learning model encrypted and integrity protected with a fresh symmetric key. We note that denial-of-service attacks are outside the threat model for this paper—the parties or the data centre may cause the computation to fail before completion. Conversely, any attempt to tamper with the enclave memory (including its code and data) would be caught as it is read by the SGX processor, and hence the job completion guarantees the integrity of the whole run. Finally, the system needs to guarantee that all

parties get access to the output. To achieve this, the enclave sends the encrypted output to every party over their secure authenticated channel, and waits for each of them to acknowledge its receipt and integrity. It then publishes the output key, sending it to all parties, as well as to any reliable third-party (to ensure its fair availability).

6 Evaluation

This section describes our experiments to evaluate the overhead of running our machine learning algorithms with privacy guarantees. We ran oblivious and non-oblivious versions of the algorithms that decrypt and process the data inside SGX enclaves, using off-the-shelf Intel Skylake processors. Our results show that, in all cases, the overhead of encryption and SGX protection was low. The oblivious version of algorithms with irregular data structures, such as matrix factorization and decision trees, adds substantial overhead, but we find that it is still several orders of magnitude better than previous work based on advanced cryptography.

6.1 Datasets

We use standard machine learning datasets from the *UCI Machine Learning Repository*.⁴ We evaluate matrix factorization on the MovieLens dataset [28]. Table 1 summarizes our datasets and configuration parameters.

The *Nursery* dataset describes the outcomes of the Slovenian nursery admission process for 12,960 applications in the 1980s. Given eight socio-economic attributes about the child and parents, the task is to classify the record into one out of five possible classes. We use the 0/1 encoding of the attributes as we evaluate the records on binary decision trees. Hence, each record in the dataset is represented using 27 features.

The *MNIST* dataset is a set of 70,000 digitized grayscale images of 28-by-28 pixels recording handwritten digits written by 500 different writers. The task is to classify each image into one of ten possible classes.

The *SUSY* dataset comprises 5,000,000 instances produced by Monte Carlo simulations of particle physics processes. The task is to classify, based on 18 observed features, whether the particles originate from a process producing supersymmetric (*SUSY*) particles or not.

The *MovieLens* datasets contain movie ratings (1–5); 100K ratings given by 943 users to 1682 movies.

The datasets were chosen either because they were used in prior work on secure ML (e.g., *Nursery* in [13], *MovieLens* in [47, 48]) or because they are one of the largest in the UCI repository (e.g., *SUSY*), or because they represent common benchmarks for particular algorithms (e.g., *MNIST* for neural networks).

Our learning algorithms are iterative—the accuracy (and execution time) of the model depends on the number

of iterations. In our experiments, we fixed the number of iterations a priori to a value that typically results in convergence: 10 for k-means, 5 for neural network, 10 for SVM, and 20 for matrix factorization.

6.2 Setup

The experiments were conducted on a single machine with quad-core Intel Skylake processor, 8GB RAM, and 256GB solid state drive running Windows 10 enterprise. This processor limits the amount of platform memory that can be reserved for enclaves to 94MB (out of a total of 128MB of protected memory). Each benchmark is compiled using the Microsoft C/C++ compiler version 17.00 with the O2 flag (optimize for speed) and linked against the Intel SGX SDK for Windows version 1.1.30214.81. We encrypted and integrity protected the input datasets with AES-GCM; we used a hardware-accelerated implementation of AES-GCM based on the Intel AES-NI instructions. We ran non-oblivious and oblivious versions of our algorithms that decrypt and process the binary data inside SGX enclaves. We compare the run times with a baseline that processes the data in plaintext and does not use SGX protection. Table 1 summarizes the relative run time for all the algorithms (we report averages over five runs). Next we analyze the results for each algorithm.

6.3 K-Means

We have implemented a streaming version of the k-means clustering algorithm to overcome space constraints of enclaves. Our implementation partitions the inputs into batches of a specified size, copies each batch into enclave memory, decrypts it and processes each point within the batch.

Table 1 shows the overheads for partitions of size 1MB. The non-oblivious and oblivious versions have overheads of 91% and 199% over baseline (6.8 seconds). The overhead for the non-oblivious version is due to the cost of copying encrypted data into the enclave and decrypting it.

We observe similar overheads for longer executions. The overheads decrease with the number of clusters (34% with 30 clusters and 11% with 50 clusters for non-oblivious version) and (154% for 30 clusters and 138% for 50 clusters for oblivious version) as the cost of input decryption is amortized over cluster computation. By comparison, recent work [38] based on cryptographic primitives reports 5 to 6 orders of magnitude slowdown for k-means.

6.4 Neural Networks

We have implemented a streaming version of the algorithm for training a convolution neural network (CNN) on top of an existing library [20]. Table 1 shows the overheads of training the network for the MNIST dataset.

⁴<https://archive.ics.uci.edu/ml/>

Algorithm	SGX+enc.	SGX+enc.+obl.	Dataset	Parameters	Input size	# Instances
K-Means	1.91	2.99	MNIST	$k=10, d=784$	128MB	70K
CNN	1.01	1.03	SUSY	$k=2, d=18$	307MB	2.25M
SVM	1.07	1.08	MovieLens	$n=943, m=1,682$	2MB	100K
Matrix fact.	1.07	115.00	Nursery	$k=5, d=27$	358KB	6.4K
Decision trees	1.22	31.10				

Table 1: Relative run times for all algorithms with SGX protection + encryption, and SGX protection + encryption + data obliviousness, compared with a baseline that processes the data in plaintext without SGX protection. Parameters of the datasets used for each algorithm are provided on the right, where d is the number of features, k is the number of classes, n is the number of users and m is the number of movies in the MovieLens dataset.

The low overheads ($< 0.3\%$) reflect the observation that the training algorithm is predominantly data oblivious, hence running obliviously does not increase execution time while achieving the same accuracy. We are aware that state-of-the-art implementations use data-dependent optimizations such as max pooling and adding noise; finding oblivious algorithms that support these optimizations with good performance remains an open problem.

6.5 SVM

As described in Section 4.2, the correctness of supervised learning methods requires that the input data instances be independent and identically distributed. Our oblivious SVM implementation achieves this by accessing a batch of l data instances uniformly at random during each iteration. We implement random access by copying the partition containing the instance into enclave memory, decrypting the partition and then accessing the data instance. In the experiments we set data partitions to be of size 2KB and $l = 20$. In addition, we use conditional move instructions to make data accesses within the training algorithm oblivious. These modifications allow us to process datasets much larger than enclave memory. Our evaluation (Table 1) shows that random access adds a 7% overhead to the non-oblivious SVM algorithm, whereas the additional overhead of the oblivious algorithm is marginal.

6.6 Decision Tree Evaluation

For the Nursery dataset, we use an offline-trained ensemble of 32 sparse decision trees (182KB) with 295–367 nodes/leaves each and depths ranging from 14 to 16 layers. For this dataset, as shown in Table 1, our oblivious classifier running inside an enclave has an average overhead of 31.1x over the baseline (255ms vs. 10ms). The oblivious implementation of the algorithm employs the `oget()` primitive (see Section 3) for all data-dependent array lookups. Without this optimization, using `omoveEx()` for the element-granular scanning of arrays instead, the overhead is much higher (142.27x on average). We observe that our oblivious implementation scales well to even very large trees. For example, for an

ensemble of 32 decision trees (16,860KB) with 30,497–32,663 nodes/leaves and 35–45 layers each,⁵ the average overhead is 63.16x over the baseline.

In comparison, prior work based on homomorphic encryption [13] uses much smaller decision trees (four nodes on four layers for the Nursery dataset), has higher overheads and communication costs, and scales poorly with increasing depth. Our experiments show that smaller depth trees have much lower accuracy (82% for depth 4 and 84% for depth 5). In contrast, our classifier for the Nursery dataset achieves an accuracy of 98.7%.

6.7 Matrix Factorization

We measure the performance of our gradient descent on the MovieLens dataset. We implemented both the baseline algorithm and the oblivious algorithm of Section 4.6. As for k-means, we stream the input data (once) into the enclave to initialize the data structures, then we operate on them in-place. We also implemented the oblivious method of Nikolaenko *et al.* [48] to compare its overhead with ours (see Section 7 for the asymptotic comparison). We did not use garbled circuits, and merely implemented their algorithm natively on Skylake CPUs.

In each experiment, we set the dimension of user and vector profiles to $d = 10$, following previous implementations in [47, 48]. We experimented with fixed numbers of iterations $T = 1, 10, 20$. With higher number of iterations the prediction rate of the model improves. For example, when using 90K instances for training, the mean squared error of prediction on 10K test dataset drops from 12.94 after 1 iteration, to 4.04 after 20 iterations, to 1.06 after 100 iterations (with $\lambda = \mu = \gamma = 0.0001$).

Table 1 reports the overheads for the MovieLens-100K dataset. The oblivious version takes 49s, versus 0.43s for the baseline, reflecting the cost of multiple oblivious sorting for each of the $T = 20$ iterations. With smaller number of iterations, the running times are 8.2s versus 0.03s for $T = 1$, and 27s versus 0.21s for $T = 10$. (As a sanity check, a naive oblivious algorithm that accesses

⁵The numbers correspond to a random forest trained on the standard *Covertype* dataset from the UCI repository.

T	This work	Previous work
1	8	14 (1.7x)
10	27	67 (2.4x)
20	49	123 (2.5x)

Table 2: Comparison of running times (in seconds) of oblivious matrix factorization methods on the MovieLens dataset: our work is the method in Section 4.6 and previous work is our implementation of an algorithm in [48] *without* garbled circuits. T is the number of algorithm iterations.

all entries in U and V to hide its random accesses runs in 1850s for $T = 10$.)

Table 2 compares the overheads of our oblivious algorithm and the one of [48]. As expected, our method outperforms theirs as the number of iterations grows, inasmuch as it sorts smaller data structures.

Comparison with cryptographic evaluations: We are aware of two prior evaluations of oblivious matrix factorization [47, 48]. Both solutions are based on garbled circuits, and exploit their parallelism. Both only perform one iteration ($T = 1$). Nikolaenko *et al.* (in 2013) report a run time of 2.9 hours for 15K ratings (extracted from 100K MovieLens dataset) using two machines with 16 cores each. Nayak *et al.* (in 2015) report a run time of 2048s for 32K ratings, using 32 processors.

6.8 Security Evaluation

We experimentally confirmed the data-obliviousness of all enclave code for each of our algorithms, as follows. We ran each algorithm in a simulated SGX environment⁶ and used Intel’s Pin framework [42] to collect runtime traces that record all memory accesses of enclave code, not only including our core algorithms, but also all standard libraries and SGX framework code. For each algorithm, we collected traces for a range of different inputs of the same size and compared code and data accesses at cache-line granularity, simulating the powerful attacker from Section 2. While we initially discovered deviations in the traces due to implementation errors in our oblivious primitives and algorithmic modifications, we can report that the final versions of all implementations produce uniform traces that depend only on the input size.

7 Related Work

Secure multi-party machine learning General cryptographic approaches to secure multi-party computation

⁶For debugging purposes, the Intel SGX SDK allows for the creation of simulated SGX enclaves. Those simulated enclaves have largely the same memory layout as regular SGX enclaves, but are not isolated from the rest of the system. In simulation mode, SGX instructions are emulated in software inside and outside the enclave with a high level of abstraction.

are based on garbled circuits, secret sharing and encryption with homomorphic properties. Lindell and Pinkas [36] survey these techniques with respect to data mining tasks including machine learning. It is worth noting that beside mathematical assumptions, some of the above approaches also rely on (a subset of) computing parties being honest when running the protocol as well as non-colluding.

Garbled circuits [71] provide a mechanism for multiple parties to compute any function on their joint inputs without having to reveal the inputs to each other. Solutions based on garbled circuits have been tailored for several specific machine learning tasks including matrix factorization [48], and training of a decision tree [6, 35]. GraphSC [47] and ObliVM [38] are two recent programming frameworks for secure computation using garbled circuits. The former framework offers a paradigm for parallel computation (e.g., MapReduce) and the latter uses a combination of ORAM and garbled circuits.

Training of an SVM kernel [34] and construction of a decision tree [18] have been proposed based on secret-sharing and oblivious transfer.

Homomorphic encryption lets multiple parties encrypt their data and request the server to compute a joint function by performing computations directly on the ciphertexts. Bost *et al.* [13] study classification over encrypted data in the model where the server performs classification by operating on semi-homomorphic encrypted data; whenever the server needs to perform operations not supported by the encryption, it engages in a protocol with a party that can decrypt the data and perform the necessary computation. Solutions based on fully-homomorphic encryption have been proposed for training several ML algorithms [27, 69] and for classifying decision trees [68].

Shokri and Shmatikov [59] describe a method for multiple parties to compute a deep neural network on joint inputs. This method does not rely on cryptographic primitives. It assumes that the parties train their own model and do not share the data with each other, but exchange intermediate parameters during training. Our model is different as parties in our solution do not perform any computation and do not learn anything about the training process; after the training they can either obtain the model, if they agreed to, or use it for querying in a black-box manner.

Privacy implications of revealing the output of a machine learning algorithm, i.e., the model, is orthogonal to the focus of this paper; we refer the reader to Fredriksson *et al.* [21, 22] on this topic. As a remedy, differential privacy guarantees for the output of several machine learning algorithms have been studied by Blum *et al.* [11].

Data-oblivious techniques Oblivious RAM (ORAM) [25] is a general protection technique against

side-channels on memory accesses. Though recent advances in this space [62] have significantly decreased the ORAM overhead, there are cases where the default solution does not always meet system requirements. First, many ORAM solutions offer a tradeoff between the size of the private memory and the overhead they incur. In current CPUs, registers act as an equivalent of processor’s private memory, however their number is limited, e.g., even for the latest x86 generations, less than 2KB can be stored in all general purpose and SIMD registers combined. Second, ORAM does not hide the number of accesses. That is, if this number depends on a sensitive input (e.g., number of movies rated by each user) then fake accesses need to be generated to hide the real number of accesses. Finally, ORAM is ideal for programs that make few accesses in a large dataset. For algorithms that process data multiple times, customized solutions often perform better (e.g., MapReduce [47, 49]). Machine learning algorithms fall in the latter category as they need all input instances to train and use the model.

Raccoon [53] and GhostRider [37] propose general compiler techniques for protecting memory accesses of a program. Some of the techniques they deploy are ORAM and execution of both branches of if-else statements. However, general techniques are less effective in cases where an algorithm accesses data in a structured way that can be exploited for greater efficiency. For example, compiling matrix factorization using these techniques is not trivial as the interleaving of the accesses between internal data structures has to be also protected. (The interleaving depends on sensitive information such as rating counts per user and per movie which have to be taken into account.)

Asymptotical comparison of individual algorithms
 We now compare the asymptotic performance of our data-oblivious algorithms to prior work. We evaluate the overhead of obtaining oblivious properties only. That is, we do not consider the cost of their secure implementation on SGX (our approach) or using garbled circuits in [38, 47, 48] (though the latter is known to add large run time overheads).

ObliVM [38] uses a streaming version of MapReduce to perform oblivious *k-means* which is then compiled to garbled circuits. The algorithm relies on oblivious sorting to update the centroids at each iteration, resulting in the running time of $O(T(nkd + dn(\log n)^2))$ (ignoring conversion to garbled circuits). Since our algorithm takes $O(Tnkd)$ time, the asymptotical comparison between the two depends on the relation between values k and $O((\log n)^2)$. Moreover, oblivious sorting incurs high constants and our experiments confirmed that our simple method was more efficient.

The algorithmic changes required to make *SVM* and *Neural Networks* oblivious can be captured with automated tools for compiling code into its oblivious counterpart. Instead of an oblivious shuffle or sort between the iterations, these methods would place input instances into an ORAM and then sample them by accessing the ORAM. Since all n instances are accessed at each iteration, the asymptotical cost of the two solutions remains the same. However, such tools either use a backend that would require careful adaption for the constrained SGX environment (for example, GhostRider [37] defines its own source language and ObliVM [38] translates code into circuits) or they are not as optimized as our approach (for example, Raccoon [53] always executes both code paths of a secret-dependent conditional statement and its described array scanning technique is less fine-tuned for the x86 architecture than ours).

Our simple data-oblivious *decision tree* algorithm is adequate for ad hoc tree evaluations, and scales up to reasonably large forests. With larger irregular data structures, algorithms based instead on oblivious data structures [38, 66] may be more effective as they store data in elaborate randomized data structures that avoid streaming over all the tree leaves. Though their asymptotical performance dominates our approach — $O((\log n)^2)$ vs. $O(n)$, assuming height of the tree of $O(\log n)$ — as pointed out in [53] ORAM-based solutions improve over the plain scanning of arrays only for larger n due to the involved constants. Moreover, private memory of size $O(\log n)$ is assumed in [66] while as mentioned earlier, private memory for SGX is limited to registers. Oblivious tree can be implemented also via ORAM with constant private memory size [33], incurring $O((\log n)^3 / \log \log n)$ overhead.

Finally, oblivious *matrix factorization* for garbled circuits, rather than SGX processors, was considered in [48] and [47]. Nikolaenko *et al.* [48] also rely on a data structure that combines both user and movie profiles: They maintain a global matrix of size $M + n + m$ with M rows for the ratings, n rows for the users, and m rows for the movies. Their updates are performed in several sequential passes, and synchronized using a sorting network on the whole data structure. Hence, their algorithm runs in time $O(T(M + n + m)(\log(M + n + m))^2)$, dominated by the cost of sorting the rows of the matrix at every iteration. GraphSC [47] implements matrix factorization using an oblivious parallel graph processing paradigm. However, this method also relies on oblivious sorting of $M + n + m$ profiles, hence, asymptotically it incurs the same cost. In comparison, our approach sorts on that scale only during Setup and, besides, those costly operations only sort user ids and ratings—not the larger profiles in \mathbb{R}^d . Then, asymptotically, our iterations are more efficient due to a smaller logarithmic factor as we

sort fewer tuples at a time: $O(T(M + \tilde{n})(\log \tilde{n})^2)$ where $\tilde{n} = \max(n, m)$. As we showed in the evaluation section our method also outperforms [48] in practice.

Similar to prior oblivious matrix factorization techniques [47, 48], our method is easily parallelizable. First, an oblivious sort that runs in time $O(n(\log n)^2)$ sequentially can run in time $O((\log n)^2)$ with n parallel processes. Besides, each row in our data structures \mathbf{U} and \mathbf{V} can be processed independently, and aggregated in time $\log(M + \tilde{n})$, as in the method described in [47]. Even with parallel processing, our method is more efficient, because the depth of the computation stays logarithmic in \tilde{n} for our method and M in theirs [47, 48].

Secure hardware TrustedDB [5], Cipherbase [3], and Monomi [64] use different forms of trusted hardware to process database queries with privacy. Haven [8] runs unmodified Windows applications in SGX enclaves, and VC3 [57] proposes a cloud data analytics framework based on SGX. None of these systems provides protection from side-channel attacks. These systems were evaluated using SGX emulators. In contrast, we are the first to evaluate implementations of machine learning algorithms on real SGX processors.

8 Conclusions

We presented a new practical system for privacy-preserving multi-party machine learning. We propose data-oblivious algorithms for support vector machines, matrix factorization, decision trees, neural networks, and k-means. Our algorithms provide strong privacy guarantees: they prevent exploitation of side channels induced by memory, disk, and network accesses. Experiments with an efficient implementation based on Intel SGX Skylake processors show that our system provides good performance on realistic datasets.

References

- [1] AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. An $O(n \log n)$ sorting network. In *ACM Symposium on Theory of Computing (STOC)* (1983).
- [2] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for CPU based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).
- [3] ARASU, A., BLANAS, S., EGURO, K., KAUSHIK, R., KOSSMANN, D., RAMAMURTHY, R., AND VENKATESAN, R. Orthogonal security with Cipherbase. In *Conference on Innovative Data Systems Research (CIDR)* (2013).
- [4] ARASU, A., AND KAUSHIK, R. Oblivious query processing. In *International Conference on Database Theory (ICDT)* (2014).
- [5] BAJAJ, S., AND SION, R. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. In *IEEE Transactions on Knowledge and Data Engineering* (2014).
- [6] BARNI, M., FAILLA, P., KOLESNIKOV, V., LAZZERETTI, R., SADEGHI, A., AND SCHNEIDER, T. Secure evaluation of private linear branching programs with medical applications. In *European Symposium on Research in Computer Security (ESORICS)* (2009).
- [7] BATCHER, K. E. Sorting networks and their applications. In *Spring Joint Computer Conf.* (1968).
- [8] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).
- [9] BELL, R. M., AND KOREN, Y. Lessons from the Netflix prize challenge. *ACM SIGKDD Explorations Newsletter* 9, 2 (2007).
- [10] BISHOP, C. M. *Neural networks for pattern recognition*. Oxford university press, 1995.
- [11] BLUM, A., DWORK, C., MCSHERRY, F., AND NISSIM, K. Practical privacy: The SuLQ framework. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)* (2005).
- [12] BOSER, B. E., GUYON, I. M., AND VAPNIK, V. N. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory* (1992).
- [13] BOST, R., POPA, R. A., TU, S., AND GOLDWASSER, S. Machine learning classification over encrypted data. In *Symposium on Network and Distributed System Security (NDSS)* (2015).
- [14] BREIMAN, L. Random forests. *Machine Learning* 45, 1 (2001).
- [15] BREIMAN, L., FRIEDMAN, J. H., OLSHEN, R. A., AND STONE, C. J. *Classification and Regression Trees*. Wadsworth, 1984.
- [16] CRIMINISI, A., SHOTTON, J., AND KONUKOGLU, E. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends in Computer Graphics and Vision* 7, 2-3 (2012).

- [17] C. YAO, A. Protocols for secure computations (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)* (1982).
- [18] DE HOOGH, S., SCHOENMAKERS, B., CHEN, P., AND OP DEN AKKER, H. Practical secure decision tree learning in a teletreatment application. In *Financial Cryptography and Data Security (FC)* (2014).
- [19] DWORK, C., MCSHERRY, F., NISSIM, K., AND SMITH, A. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)* (2006).
- [20] Fast CNN library. <http://fastcnn.codeplex.com/> (accessed 17/02/2016).
- [21] FREDRIKSON, M., JHA, S., AND RISTENPART, T. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM Conference on Computer and Communications Security (CCS)* (2015).
- [22] FREDRIKSON, M., LANTZ, E., JHA, S., LIN, S., PAGE, D., AND RISTENPART, T. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium* (2014).
- [23] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)* (2009).
- [24] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game. In *ACM Symposium on Theory of Computing (STOC)* (1987).
- [25] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996).
- [26] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. Deep learning. Book in preparation for MIT Press, 2016.
- [27] GRAEPEL, T., LAUTER, K., AND NAEHRIG, M. ML confidential: Machine learning on encrypted data. In *International Conference on Information Security and Cryptology (ICISC)* (2013).
- [28] HARPER, F. M., AND KONSTAN, J. A. The MovieLens datasets: History and context. In *ACM Transactions on Interactive Intelligent Systems (TiS)* (2015).
- [29] HOEKSTRA, M., LAL, R., PAPPACHAN, P., ROZAS, C., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).
- [30] HYEONG HONG, J., AND FITZGIBBON, A. Secrets of matrix factorization: Approximations, numerics, manifold optimization and random restarts. In *Proceedings of the IEEE International Conference on Computer Vision* (2015).
- [31] INTEL CORP. Intel 64 and IA-32 architectures software developer’s manual—combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c, 2013. No. 325462-048.
- [32] KOREN, Y., BELL, R., AND VOLINSKY, C. Matrix factorization techniques for recommender systems. *Computer*, 8 (2009).
- [33] KUSHILEVITZ, E., LU, S., AND OSTROVSKY, R. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2012).
- [34] LAUR, S., LIPMAA, H., AND MIELIKÄINEN, T. Cryptographically private support vector machines. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2006).
- [35] LINDELL, Y., AND PINKAS, B. Privacy preserving data mining. *Journal of Cryptology* (2000).
- [36] LINDELL, Y., AND PINKAS, B. Secure multi-party computation for privacy-preserving data mining. *IACR Cryptology ePrint Archive* (2008).
- [37] LIU, C., HARRIS, A., MAAS, M., HICKS, M. W., TIWARI, M., AND SHI, E. Ghostrider: A hardware-software system for memory trace oblivious computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).
- [38] LIU, C., WANG, X. S., NAYAK, K., HUANG, Y., AND SHI, E. ObliVM: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy (S&P)* (2015).
- [39] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P)* (2015).
- [40] LLOYD, S. P. Least squares quantization in PCM’S. *Bell Telephone Labs Memo* (1957).
- [41] LLOYD, S. P. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982).

- [42] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2005).
- [43] MACQUEEN, J. Some methods for classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematics, Statistics and Probability, Vol. 1* (1967).
- [44] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay: a secure two party computation system. In *USENIX Security Symposium* (2004).
- [45] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).
- [46] NARAYANAN, A., AND SHMATIKOV, V. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy (S&P)* (2008).
- [47] NAYAK, K., WANG, X. S., IOANNIDIS, S., WEINSBERG, U., TAFT, N., AND SHI, E. GraphSC: Parallel secure computation made easy. In *IEEE Symposium on Security and Privacy (S&P)* (2015).
- [48] NIKOLAENKO, V., IOANNIDIS, S., WEINSBERG, U., JOYE, M., TAFT, N., AND BONEH, D. Privacy-preserving matrix factorization. In *ACM Conference on Computer and Communications Security (CCS)* (2013).
- [49] OHRIMENKO, O., COSTA, M., FOURNET, C., GKANTSIDES, C., KOHLWEISS, M., AND SHARMA, D. Observing and preventing leakage in MapReduce. In *ACM Conference on Computer and Communications Security (CCS)* (2015).
- [50] OHRIMENKO, O., GOODRICH, M. T., TAMASSIA, R., AND UPFAL, E. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages and Programming (ICALP)*, vol. 8573. Springer, 2014.
- [51] QUINLAN, J. R. Induction of decision trees. *Machine Learning* 1, 1 (1986).
- [52] QUINLAN, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
- [53] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium* (2015).
- [54] RASTOGI, A., HAMMER, M. A., AND HICKS, M. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE Symposium on Security and Privacy (S&P)* (2014).
- [55] SARWAR, B., KARYPIS, G., KONSTAN, J., AND RIEDL, J. Application of dimensionality reduction in recommender system – A case study. Tech. rep., DTIC Document, 2000.
- [56] SCHÖLKOPF, B., AND SMOLA, A. J. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.
- [57] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDES, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy data analytics in the cloud using sgx. In *IEEE Symposium on Security and Privacy (S&P)* (2015).
- [58] SHALEV-SHWARTZ, S., SINGER, Y., SREBRO, N., AND COTTER, A. Pegasos: Primal estimated sub-gradient solver for SVM. *Mathematical programming* 127, 1 (2011).
- [59] SHOKRI, R., AND SHMATIKOV, V. Privacy-preserving deep learning. In *ACM Conference on Computer and Communications Security (CCS)* (2015).
- [60] SINHA, R., COSTA, M., LAL, A., LOPES, N., SE-SHIA, S., RAJAMANI, S., AND VASWANI, K. A design and verification methodology for secure isolated regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2016).
- [61] SMOLA, A. J., AND SCHÖLKOPF, B. A tutorial on support vector regression. *Statistics and computing* 14, 3 (2004).
- [62] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C. W., REN, L., YU, X., AND DEVADAS, S. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM Conference on Computer and Communications Security (CCS)* (2013).
- [63] TSOCHANARIDIS, I., JOACHIMS, T., HOFMANN, T., AND ALTUN, Y. Large margin methods for structured and interdependent output variables. In *Journal of Machine Learning Research* (2005).

- [64] TU, S., KAASHOEK, M. F., MADDEN, S., AND ZELDOVICH, N. Processing analytical queries over encrypted data. In *International Conference on Very Large Data Bases (VLDB)* (2013).
- [65] VAPNIK, V. N., AND VAPNIK, V. *Statistical learning theory*, vol. 1. Wiley New York, 1998.
- [66] WANG, X. S., NAYAK, K., LIU, C., CHAN, T., SHI, E., STEFANOV, E., AND HUANG, Y. Oblivious data structures. In *ACM Conference on Computer and Communications Security (CCS)* (2014).
- [67] WESTON, J., AND WATKINS, C. Support vector machines for multi-class pattern recognition. In *ESANN* (1999).
- [68] WU, D. J., FENG, T., NAEHRIG, M., AND LAUTER, K. Privately evaluating decision trees and random forests. *IACR Cryptology ePrint Archive* (2015).
- [69] XIE, P., BILENKO, M., FINLEY, T., GILAD-BACHRACH, R., LAUTER, K. E., AND NAEHRIG, M. Crypto-nets: Neural networks over encrypted data. *CorR abs/1412.6181* (2014).
- [70] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P)* (2015).
- [71] YAO, A. C. How to generate and exchange secrets (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)* (1986).

A Illustration of Oblivious Matrix Factorization from Section 4.6

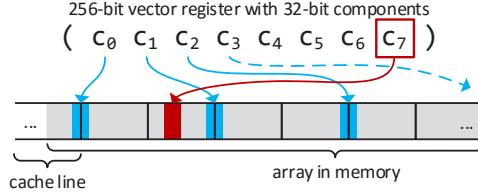
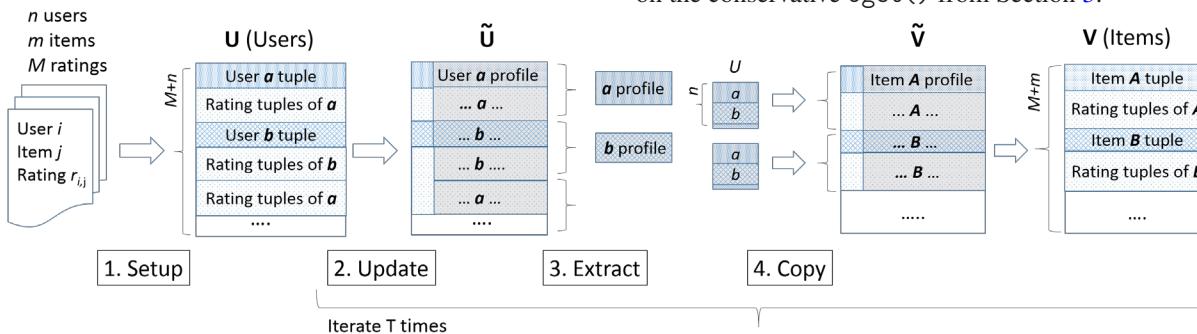


Figure 4: Optimized array scanning using the 256-bit vector instruction vpgatherdd

B Optimized Array Scanning

The `oget()` primitive can be further optimized using `vpgatherdd` as follows. We make sure that a certain number of components of the vector register load values that span *two* cache lines. (This can be done by loading two bytes from one cache line and two bytes from the next one, recall that each component loads 4 bytes.) Hence, up to 16 cache lines can potentially be touched with a single instruction.

We assign components to cache lines in a careful manner. The first few components request addresses within dummy cache lines or cache lines that contain the values of interest (whose addresses should be kept secret). The values of interest are loaded into the remaining components. The concept is depicted in Figure 4 where components C_0 and C_2 - C_6 request dummy cache lines, C_1 requests the cache lines that contain the desired value which is loaded into C_7 . In this configuration, four bytes are read obliviously from a memory region of size $7 \cdot 2 \cdot 64$ bytes = 896 bytes with a single `vpgatherdd` instruction. The method easily generalizes when more bytes (e.g., 8 bytes using C_6 and C_7) are to be read.

This technique can significantly increase throughput (up to 2x in some micro-benchmarks outside enclaves on recent Intel Skylake processors). However, it requires that `vpgatherdd` appears as a truly atomic operation or, at least, that the hardware loads dummy components before secret ones (and those are then loaded from the cache). Though this may be true in a software-only attacker model, it is not the case in the powerful threat model in Section 2. Hence, our implementation relies on the conservative `oget()` from Section 3.

Thoth: Comprehensive Policy Compliance in Data Retrieval Systems

Eslam Elnikety

Aastha Mehta

Anjo Vahldiek-Oberwagner
Peter Druschel

Deepak Garg

Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Data retrieval systems process data from many sources, each subject to its own data use policy. Ensuring compliance with these policies despite bugs, misconfiguration, or operator error in a large, complex, and fast evolving system is a major challenge. Thoth provides an efficient, kernel-level compliance layer for data use policies. Declarative policies are attached to the systems' input and output files, key-value tuples, and network connections, and specify the data's integrity and confidentiality requirements. Thoth tracks the flow of data through the system, and enforces policy regardless of bugs, misconfigurations, compromises in application code, or actions by unprivileged operators. Thoth requires minimal changes to an existing system and has modest overhead, as we show using a prototype Thoth-enabled data retrieval system based on the popular Apache Lucene.

1 Introduction

Online data retrieval systems typically serve a searchable corpus of documents, web pages, blogs, personal emails, online social network (OSN) profiles and posts, along with real-time microblogs, stock and news tickers. Examples include large providers like Amazon, Facebook, eBay, Google, and Microsoft, and also numerous smaller, domain-specific sharing, trading and networking sites run by organizations, enterprises, and governments.

Each data item served or used by a retrieval system may have its own usage policy. For instance, email is private to its sender/receiver(s), OSN data and blogs limited to friends, and corporate documents limited to employees. External data stream providers may restrict the use of (meta)data, and require expiration. The provider's privacy policy may require that a user's query and click stream be used only for personalization. Lastly, providers must comply with local laws, which may require them, for instance, to filter certain data items within a given jurisdiction.

Ensuring compliance with applicable policies is labor-intensive and error-prone [36]. The policy actually in effect for a data item may depend on checks and settings in

many components and several layers of a system, making it difficult to audit and reason about. Moreover, any bug, misconfiguration, or compromise in a large and evolving application codebase could violate a policy. The problem affects both large providers with complex, fast evolving systems and smaller providers with limited IT budgets. Indeed, reports of data losses abound [14, 1, 44, 11, 13]. The stakes are high: providers stand to lose customer confidence, business and reputation, and may face fines. Hence, developing technical mechanisms to enforce policies in data retrieval systems is important. In fact, the Grok system combines lightweight static analysis with heuristics to annotate source code to check for policy violations in Bing's back-end [36].

Existing policy compliance systems for data retrieval, including Grok, usually target *column-specific policies*—policies that apply uniformly to all data of a specific type, e.g., the policy “no IP address can be used for advertising.” However, no existing work covers the equally important *individual policies* that are specific to individual data items or to a given client's data items. For example, Alice's blog posts, but not Bob's, may be subject to the policy “visible only to Alice's friends”. Similarly, the expiration time of every item in a news ticker may be different. In fact, all policies mentioned a couple of paragraphs ago are individual policies. It is this (significant and important) missing part of policy enforcement that we wish to address in this paper. Specifically, we present Thoth, a policy compliance layer integrated into the Linux kernel to enforce both individual and column-specific policies efficiently.

We briefly describe the key insights in Thoth's design. First, by design, Thoth separates policies from application code. A policy specifying confidentiality and integrity requirements may be associated with any data conduit, i.e, a file, key-value tuple, named pipe or network connection, and is enforced on all application code that accesses the conduit's data or data derived from that data. Thoth provides a declarative language for specifying policies. The language itself is novel; in addition to standard access (read/write) policies, it also allows specifying data declassification policies by stipulating how

access policies may change along a data flow.

Second, unlike column-specific policies, individual policies may not be very amenable to static analysis because a given program variable may contain data with very different individual policies over time at the same program point and, hence, the abstraction of static analysis may lose precision quickly. So, Thoth uses dynamic analysis. It intercepts I/O in the kernel, tracks the flow of data at the granularity of conduits and processes (similar to Flume [28]), and enforces policies at process boundaries. This incurs a runtime overhead but we show that the overhead is not too high. With an optimized prototype implementation, we measure an overhead of 0.7% on indexing and 3.6% on query throughput in the widely used search engine Apache Lucene. While this overhead may be too high for large-scale data retrieval systems, we believe that it can be optimized further and that it is already suitable for domain-specific, medium-scale data retrieval systems run by organizations, enterprises and governments. Moreover, application code requires very few changes to run with Thoth (50 lines in a codebase of 300,000 LoC in our experiments).

Third, the complexity of a data retrieval system often necessitates some declassification to maintain functionality. For instance, a search process that consults an index computed over a corpus containing the private data of more than one individual cannot produce any readable results without declassification. To handle this and similar situations, we introduce a new form of declassification called *typed declassification*, which allows the declassification of data in specific forms (types). To accommodate the aforementioned search process, all source data policies allow declassification into a list of search results (document names). Hence, the search process can function as usual. At the same time, the possibility of data leaks is limited to a very narrow channel: To leak information from a private file, the search process' code must maliciously encode the information in a list of valid document names. Given that the provider has a genuine interest in preventing data breaches and that the search process is an internal component that is unlikely to be compromised in a casual external attack, the chance of having such malicious code in the search process is low. Thus, typed declassification is a pragmatic design point in the security-functionality trade-off for our threat model. Note that typed declassification needs content-dependent policies, which our policy language supports.

To summarize, the contributions of this work are: (1) A policy language that can express individual access and declassification policies declaratively (Section 2); (2) the design of a kernel-level monitor to enforce policies by I/O interception and lightweight taint propagation (Section 3); (3) application of the design to medium-scale data retrieval systems, specifically

Apache's Lucene (Sections 2; 5); and (4) an optimized prototype implementation and experimental evaluation to measure overheads (Sections 4, 6).

2 Thoth policies

Thoth is a kernel-level policy compliance layer that helps data retrieval system providers enforce confidentiality and integrity policies on the data they collect and serve. In Thoth, the *provider* attaches policies to data sources (documents and live streams, posts and profiles, user click history, etc.) based on the privacy preferences of clients, external (e.g., legal) and internal usage requirements. Thoth *tracks data flows* by intercepting all IPC and I/O in the kernel, and it propagates source policies along these flows. It enforces policy conditions when data leaves the system, or when a declassification happens. The policy attached to a data source is a *complete, one point description* of all privacy and integrity rules in effect for that source.

Thoth policies are specified in a new, expressive declarative language, separate from application code. In this section, we describe this policy language briefly, discuss example policies that clients, data sources, and the provider might wish to enforce in a data retrieval system, and give a glimpse of how to express these policies in Thoth's policy language. More policy examples are included in Appendix A. Section 3 explains how Thoth enforces these policies. We note that our policy language and enforcement are general and apply beyond data retrieval systems.

Policy language overview A Thoth policy can be attached to any *conduit*—a file, key-value tuple, named pipe or network socket that stores data or carries data in transit. The policy on a conduit protects the confidentiality and integrity of the data in the conduit and is specified in two layers. The first layer, an *access control policy*, specifies which principals may **read** and **update** the conduit and under what conditions (e.g., only before or only after a certain date). A second layer protects data *derived* from the conduit by restricting the policies of conduits downstream in the data pipeline. This layer can **declassify** data by allowing the access policies downstream to be relaxed progressively, as more and more declassification conditions are met. The second layer that specifies declassification by controlling downstream policies is the language's key novelty.¹ Another noteworthy feature is that we allow policy evaluation to depend on a conduit's state—both its data and its metadata. This allows expressing content-dependent policies and, in particular, a kind of declassification that we call *typed declassification*.

¹Our full language also supports provenance policies in the second layer by allowing control over upstream policies. Due to lack of space, we omit provenance policies here.

Arithmetic/string	Conduit	Content
add(x,y,z) $x=y+z$	cNamels(x)	x is the conduit pathname
sub(x,y,z) $x=y-z$	clIdls(x)	x is the conduit id
mul(x,y,z) $x=y*z$	clExists(x)	x is a valid conduit id
div(x,y,z) $x=y/z$	cCurrLenls(x)	x is the conduit length
rem(x,y,z) $x=y\%z$	cNewLenls(x)	x is the new conduit length
concat(x,y) $x \parallel y$	hasPol(c, p)	p is conduit c's policy
vType(x,y) is x of type y?	clsIntrinsic	does this conduit connect two confined processes?
Relational	Session	Declassification rules
eq(x,y) $x=y$	sKeyls(x)	x is the session's authentication key
neq(x,y) $x \neq y$	slpls(x)	x is the session's source IP address
lt(x,y) $x < y$		
gt(x,y) $x > y$	lpPrefix(x,y)	x is IP prefix of y
le(x,y) $x \leq y$	timels(t)	t is the current time
ge(x,y) $x \geq y$		

Table 1: Thoth policy language predicates and connectives

Layer 1: Access policies The first layer of a conduit’s policy contains two rules that specify who can read and update the conduit’s state under what conditions. We write both rules in the syntax of Datalog, which has been used widely in the past for the declarative specification of access policies [18, 20, 30]. Briefly, the read rule has the form (**read** :- cond) and means that the conduit can be read if the condition “cond” is satisfied. The condition “cond” consists of *predicates* connected with conjunction (“and”, written \wedge) and disjunction (“or”, written \vee). All supported predicates are listed in Table 1. Similarly, the update rule has the form (**update** :- cond).

Example (Client policies) Consider a search engine that indexes clients’ private data. A relevant security goal might be that a client Alice’s private emails and profile should be visible only to Alice, and only she should be able to modify this data. This *private data* policy can be expressed by attaching to each conduit holding Alice’s private items **read** and **update** rules that allow these operations only in the context of a session authenticated with Alice’s key. The latter condition can be expressed using a single predicate **sKeyls**(k_{Alice}), which means that the active session is authenticated with Alice’s public key, denoted k_{Alice} . Hence, the read rule would be **read** :- **sKeyls**(k_{Alice}). The update rule would be **update** :- **sKeyls**(k_{Alice}). (Clients, or processes running on behalf of clients, authenticate directly to Thoth, so Thoth does not rely on untrusted applications for session authentication information.)

Alice’s *friends only* blog and OSN profile should be readable by her friends as well, which can be expressed with an additional disjunctive clause in the read rule:

```
read :- sKeyls( $k_{Alice}$ )  $\vee$ 
      (sKeyls(K)  $\wedge$  (“Alice.acl”, Offset) says isFriend(K))
```

The part after the \vee is read as “the key K that authenti-

cated the current session exists in Alice.acl at some offset $Offset$.” Here, Alice.acl is a trusted key-value tuple that contains Alice’s friend list.

Following standard Datalog convention, terms like K and $Offset$ that start with uppercase letters are existentially quantified variables. The predicate **sKeyls**(K) binds K to the key that authenticates the session. During each policy evaluation, application code is expected to provide a binding for the variable $Offset$ that refers to a location in the tuple’s value saying that K belongs to a friend of Alice. Note that policy compliance does not depend on application correctness: if the application does not provide a correct offset, access will be denied.

Extending further, visibility to Alice’s *friends of friends* can be allowed by modifying the read rule to check that Alice and the owner of the current session’s key have a common friend. Then, the application code would be expected to provide an offset in Alice’s acl where the common friend exists and an offset in the common friend’s acl where the current session’s key exists.

Layer 2: Declassification policies The second layer of a conduit’s policy contains a single rule that controls the policies of downstream conduits. This rule is written (**declassify** :- cond), where “cond” is a condition or predicate on all *downstream sequences of conduits*. For instance, “cond” may say that in any downstream sequence of conduits, the access policies must allow read access only to Alice, until the calendar year is at least 2017, after which the policies may allow read access to anyone. This represents the declassification policy “private to Alice until the end of 2016”.

We represent such declassification policies using the notation of *linear temporal logic* (LTL), a well-known syntax to represent predicates that change over time [32]. We allow one new connective in “cond” in the **declassify** rule: c_1 until c_2 , which means that condition c_1 must

hold of all downstream conduits until condition c2 holds. Also, we allow a new predicate `isAsRestrictive(p1, p2)`, which checks that policy `p1` is at least as restrictive as `p2`. The two together can represent expressive declassification policies, as we illustrate next.

Example (Index policy) In the last example, we discussed confidentiality policies that reflect data owners' privacy choices. For the retrieval system to do its job, however, the input data policies must allow some declassification. Without it, the search engine, which consults an index computed over the entire corpus, including the private data of several individuals, would not be allowed to produce any readable output. We rely on the policy language's novel ability to refer to a conduit's (meta-)data to allow the selective, *typed declassification* of search results. The policy can be implemented by adding the following `declassify` rule to all searchable input data:

```
declassify :- isAsRestrictive(read, this.read) until
ONLY_CND_IDS
```

This policy stipulates that data derived from Alice's data can be written into conduits whose read rule is at least as restrictive as Alice's (which is bound to `this.read`), until it is written into a conduit which satisfies the condition `ONLY_CND_IDS`. This macro stipulates that only a list of valid conduit ids has been written. The macro expands to

```
cCurrLenIs(CurrLen) ∧ cNewLenIs(NewLen) ∧
each in(this, CurrLen, NewLen) says(CndId)
{cldExists(CndId)}
```

and permits the declassification of a list of proper conduit ids. A conduit id is a unique identifier for a conduit (conduit ids are defined in Section 3). The predicate “`each in () says () {}`” iterates over the sequence of tuples in the newly written data and checks that each is a valid conduit id. By including this declassification rule in her data item's policy, Alice allows the search engine to index her item and include it in search results. To view the contents, of course, a querier still has to satisfy each conduit's confidentiality policy.²

So far, we have assumed that the conduit ids (i.e., the names of indexed files) are not themselves confidential. If the conduit ids are themselves confidential, then the

²Our declassification policies can be intuitively viewed as state machines whose states are access policies and whose transitions are events in the data flow. For instance, the declassification policy just described is a two state machine, whose initial state has a read policy as restrictive as Alice's, and whose second state allows read access to everyone. The transition from the first to the second state is allowed when data passes through a conduit that satisfies `ONLY_CND_IDS`. This state-machine view of our policies is universal because it is well known that all LTL formulas can be represented as Büchi automata.

above `declassify` rule is insufficient since it stipulates no restriction on policies after `ONLY_CND_IDS` holds. Thus, a more restrictive `declassify` rule is needed. Ideally, we want that the read and declassify rules of the conduit that contains the list of conduit ids be at least as restrictive as the read and declassify rules of all conduits in the list. This can be accomplished by the following replacement for `ONLY_CND_IDS`.

```
cCurrLenIs(CurrLen) ∧ cNewLenIs(NewLen) ∧
each in(this, CurrLen, NewLen) willsay(CndId) ∧
{cldExists(CndId) ∧ hasPol(CndId, P) ∧
isAsRestrictive(read, P.read) ∧
isAsRestrictive(declassify, P.declassify)}
```

The predicate `hasPol(CndId, P)` binds `P` to the policy of the conduit `CndId`, and the predicates `isAsRestrictive(read, P.read)` and `isAsRestrictive(declassify, P.declassify)` enforce that the read and declassify rules of the search results are at least as restrictive as those of `CndId`. We call this modified macro `ONLY_CND_IDS+`.

Other data retrieval policies

We briefly describe several other policies relevant to data retrieval systems that we have represented in our policy language and implemented in our prototype. For the formal encodings of these policies, see Appendix A.

Data analytics Many retrieval systems transform logs of user activity into a user preferences vector, which is used for targeting ads, computing user profiles, and providing recommendations. Raw logs of user clicks and queries are typically private, so a profile vector derived from them cannot be used for any of these purposes without a declassification. A policy that allows typed declassification into a vector of a fixed size can be attached to raw user logs to ensure that the raw logs cannot be leaked from the system, but that the profile vector can be used for the above-mentioned purposes.

Provider policies The provider may need to censor certain documents when a query arrives from a particular country. For this purpose, the system uses a map of IP address prefixes to countries. Separately, the provider maintains a per-country blacklist, containing a list of censored conduit ids. The *censorship policy* takes the form of a common declassification rule on source files. The rule requires that, at a conduit connecting to a client, the client's IP prefix is looked up in the prefix map, and the corresponding blacklist is checked to see if any of the search results are censored. Both the prefix map and the blacklist are maintained in sorted order for efficient lookup. The sort order is enforced by an integrity policy on the conduits.

A second common provider policy allows employees to access client's private data for troubleshooting pur-

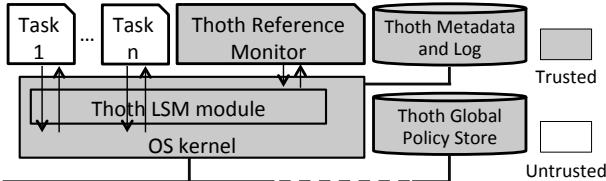


Figure 1: Thoth architecture

poses, as long as such accesses are logged for auditing. A *mandatory access logging (MAL)* policy can be added for this purpose. The policy allows accesses by authorized employees, if and only if an entry exists in a separate log file, which states a signature by the employee, the conduit being accessed, and a timestamp. The log file itself has an integrity policy that allows appends only, thus ensuring that an entry cannot be removed or overwritten. Finally, data sources must consent to provider access by allowing declassification into a conduit readable by authorized employees subject to MAL.

3 Thoth architecture and design

3.1 Overview

Figure 1 depicts the Thoth architecture. At each participating node, Thoth comprises a kernel module that intercepts I/O, a trusted reference monitor process that maintains per-task taint sets³ and evaluates policies, a persistent store for metadata and transaction log, and a persistent policy store. Each node tracks taint and enforces policies independently of other nodes. The policy store is accessible exclusively by the reference monitors and provides a consistent view of all policies. This can be attained by using either central storage for policies or a consensus protocol like Paxos [29].

Figure 2 shows the data flow model of a Thoth-protected system. An application consists of a set of tasks (i.e., processes) that execute on one or more nodes. Data flows among the tasks via *conduits*. A file, named pipe or a tuple in a key-value store is a conduit. A network connection or a named pipe is a pair of conduits, one for each direction of data traffic. Thoth identifies each conduit with a unique numeric identifier, called the conduit id. The conduit id is the hash of the path name in case of a file or named pipe, the hash of the 5-tuple $\langle \text{srcIP}, \text{srcPort}, \text{protocol}, \text{destIP}, \text{destPort} \rangle$ in case of a network connection, or the key in case of a key-value tuple. Any conduit may have an associated policy.⁴

The core of the application system is a set of *CONFINED* tasks within Thoth’s *confinement boundary*. The system interacts with the outside world via conduits (typ-

ically network connections) to external, *UNCONFINED* tasks. *UNCONFINED* tasks represent external users or components. Neither type of task is trusted by Thoth, although an *UNCONFINED* task may represent a user and may possess the user’s authentication credentials.

Policies on inbound and outbound conduits that cross the confinement boundary represent the ingress and egress policies, respectively. The read and declassification rules of an ingress policy control how data can be used and disseminated by the system whereas the update rule of an ingress policy determines who may feed data into the system. The read rule of an egress policy defines who outside the system may read the output data.

3.2 Threat model

The Thoth kernel module and reference monitor, as well as the Linux system and policy store they depend on, are trusted. Active attacks on these components are out of scope. We assume that correct policies are installed on ingress and egress conduits. In our current prototype, storage systems that hold application data are assumed to be trusted. This assumption can be relaxed by encrypting and checksumming application data in the Thoth kernel module.

Thoth makes no assumptions about the nature of bugs and misconfigurations in application components, the type of errors committed by unprivileged operators, or errors in policies on internal conduits. Subject to this threat model, Thoth provably enforces all ingress policies. In information flow control terms, Thoth can control both explicit and implicit flows, but leaks due to covert and side-channels are out of scope.

Justification Trusting the Thoth kernel module, reference monitor, and the Linux system they depend on is reasonable in practice because (i) reputable providers will install security patches on the OS and Thoth components, and correct policies; (ii) OS and Thoth are maintained by a small team of experts and are more stable than applications; thus, an attacker will likely find it more difficult to find a vulnerability in the OS or Thoth than in a rapidly evolving application with a large attack surface.

Typed declassification policies admit limited information flows, which can be exploited by malicious applications covertly. For instance, malware injected into a search engine can encode private information in the set of conduit ids it produces, if the conduits in the set themselves are public. This channel is out of scope. In practice, such attacks require significant sophistication. A successful attack must inject code strategically into the data flow before a declassification point and encode private data on a policy-compliant flow.

On the other hand, Thoth prevents the large class of practical attacks that involve direct flows to unauthorized parties, and accidental policy violations due to applica-

³A task’s taint set is the set of policies of conduits it has read.

⁴If a file has multiple hard links, each of its path names can be associated with a different policy. When a path name is used to access the file, that path name’s policies are checked.

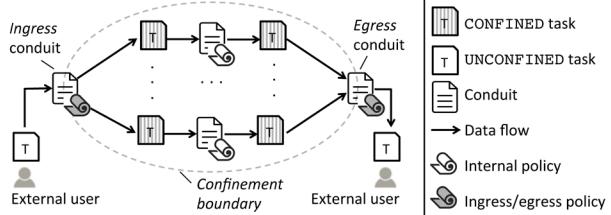


Figure 2: Thoth data flow

tion bugs, misconfigurations, and errors by unprivileged operators. We demonstrate this in Section 6.3 where a Thoth compliant search engine is able to enforce data policies, preventing (real and synthetic) bugs and misconfigurations from leaking information.

3.3 Data flow tracking and enforcement

Tracking data flow Thoth tracks data flows coarsely at the task-level. **CONFINED** and **UNCONFINED** tasks are subject to different policy checks. A **CONFINED** task may read any conduit, irrespective of the conduit’s **read** rule, but Thoth enforces each such conduit’s **declassify** rule when the task writes to other conduits. To do this, Thoth maintains the **declassify** rules of conduits read by each **CONFINED** task in the task’s metadata (these rules constitute the *taint set* of the task).

UNCONFINED tasks form the ingress and egress points for Thoth’s flow tracking; they are subject to access control checks, not tainting. An **UNCONFINED** task may read from (write to) a conduit only if the conduit’s **read** (**update**) rule is satisfied. For example, to read Alice’s private data, an **UNCONFINED** task must authenticate with Alice’s credentials. Conduits without policies can be read and written by all tasks freely.

In summary, Thoth tracks data flows across **CONFINED** tasks coarsely, and enforces declassification policies on these flows. At the ingress and egress tasks (**UNCONFINED** tasks), Thoth imposes access control through the **read** and **update** rules. Every new task starts **UNCONFINED**. The task may transition to the **CONFINED** state through a designated Thoth API call. The reverse transition is disallowed to prevent a task from reading private data in the **CONFINED** state and leaking the data to a conduit without any policy protection after transitioning to the **UNCONFINED** state.

Conduit interceptors The Thoth kernel component includes a conduit interceptor (CI) for each type of conduit. A CI for a given conduit type intercepts system calls that access or manipulate conduits of that type, and associates a conduit with its policy. Thoth has built-in CIs for kernel-defined conduit types, namely files, named pipes, and network connections. CIs for additional conduit types can be plugged in. For instance, our prototype uses a CI for the memcached key-value store (KV).

Question: Should a conduit read or write be allowed?

Inputs: t , the task reading or writing the conduit
 f , the conduit being read or written
 op , the operation being performed (read or write)

Output: Allow or deny the access

Side-effects: May update the taint set of t

```

1 if  $t$  is UNCONFINED:
2   if  $op$  is read:
3     Check  $f$ ’s read rule.
4   if  $op$  is write:
5     Check  $f$ ’s update rule.

6 if  $t$  is CONFINED:
7   if  $op$  is read:
8     Add  $f$ ’s policy to  $t$ ’s taint set.
9   if  $op$  is write:
10    // Enforce declassification policies of  $t$ ’s taint set
11    for each declassification rule  $(c \text{ until } c')$  in  $t$ ’s taint set:
12      Check that EITHER  $c'$  holds OR  $(c \text{ holds AND } f$ ’s declassification policy implies  $(c \text{ until } c'))$ .

```

Figure 3: Thoth policy enforcement algorithm

The CIs for files and named pipes associate a policy with the unique pathname of a file or pipe. The socket CI associates a policy with the network connection’s 5-tuple $\langle \text{srcIP}, \text{srcPort}, \text{protocol}, \text{destIP}, \text{destPort} \rangle$. The 5-tuple may be underspecified. For instance, the policy associated with $\langle ?, ?, ?, ?, \text{destIP}, \text{destPort} \rangle$ applies to any network connection with the specified destination IP address and port. Both ends of a network connection have the same policy. The KV CI associates a policy with a tuple’s key. The KV CI can automatically derive policies from policy templates that cover a subspace of keys (e.g., all keys with prefix `#user_profile`). It can also replace template variables with metadata, e.g., the time at which the key was created.

Policy enforcement algorithm Figure 3 summarizes the abstract checks that Thoth makes when it intercepts a conduit access. If the calling task is **UNCONFINED**, then Thoth evaluates the read or update policy of the conduit (lines 1–5). If the calling task is **CONFINED** and the operation is a read, then Thoth adds the policy of the conduit being read to the taint set of the calling task. No policy check is performed in this case (lines 6–8). To reduce the size of a **CONFINED** task’s taint set, our prototype performs *taint compression* when possible: A policy is not added if the taint set already includes an equally or more restrictive policy.

When a **CONFINED** task t writes a conduit f , there is a potential data flow from every conduit that t has read in the past to f . Hence, all declassification rules in t ’s taint set are enforced (lines 11–12). Suppose $(c \text{ until } c')$ is a **declassification** rule in t ’s taint set. Since this rule means

that condition c must continue to hold downstream *until* the declassification condition c' holds, this rule can be satisfied in one of two ways: Either the declassification condition c' holds now, or c holds now and the next downstream conduit (f here) continues to enforce (c until c'). Line 12 makes exactly this check.

End-to-end correctness of policy enforcement
 Within Thoth’s threat model, the checks described above enforce all policies on conduits and, specifically, all ingress policies. Incorrect policy configuration on internal conduits cannot cause violation of ingress policies but may cause compliant data flows to be denied by the Thoth reference monitor. Informally, this holds because our checks ensure that the conditions in every declassification policy are propagated downstream until they are satisfied.⁵

Policy comparison Thoth compares policies for restrictiveness in three cases: for taint compression, when evaluating the predicate `isAsRestrictive()`, and in line 12 of the enforcement algorithm (Figure 3). The general comparison problem is undecidable for first-order logic, so Thoth uses the following heuristics: 1) *Equality*: Compare the hashes of the two policies. 2) *Inclusion*: Check that all predicates in the less restrictive policy also appear in the more restrictive one, taking into account variable renaming and conjunctions and disjunctions between the predicates. Inclusion has exponential time complexity in the worst case, but is fast in practice. 3) *Partial evaluation*: Evaluate and delete an application-specified subpart of each policy, then try equality and inclusion. These heuristics suffice in all cases we have encountered.

Note that a policy comparison failure can never affect Thoth’s safety. However, a failure can (a) defeat taint compression and therefore increase taint size and policy evaluation overhead; or (b) cause a compliant data flow to be denied. In the latter case, a policy designer may re-state a policy so that the policy comparison succeeds.

3.4 Thoth API

Table 2 lists Thoth API functions provided to user-level tasks by means of a new system call. To check structural properties of written data (e.g., that the data is a list of conduit ids), it is often necessary to evaluate the `update` rule atomically on a *batch* of writes. Hence, Thoth supports write transactions on conduits. By default, a transaction starts with a POSIX `open()` call and ends with the `close()` call on a conduit. This behavior can be overridden by passing additional flags to `open()` and `close()`. Transactions can also be explicitly started and ended using the Thoth API calls `open_tx` and `close_tx`.

⁵A formal proof of this fact is the subject of a forthcoming paper. Our formal model and implementation support nested uses of the `until` operator, which we omitted here.

During a transaction, Thoth buffers writes in a persistent re-do log. When the transaction is closed by the application, Thoth makes the policy checks described in Figure 3. If the policy checks succeed, then the writes are sent to the conduit, else the writes are discarded. The re-do log allows recovery from crashes and avoids expensive filesystem syncs when a transaction commits.

Summary Thoth enforces ingress and egress policies despite application-level bugs, misconfigurations, and compromises, or actions by unprivileged operators. A data source’s policy specifies both access and declassification conditions and describes the source’s allowed uses completely. Thoth uses policies as taint, which differs significantly from the standard information flow control practice of using abstract labels as taint. That practice requires trusted application processes to declassify data and to control access at system edges. In contrast, Thoth relies entirely on its reference monitor for all access and declassification checks, and no application processes have to be trusted.

4 Thoth prototype

Our prototype consists of a Linux kernel module that plugs into the Linux Security Module (LSM) interface, and a reference monitor. We also changed a few (22) lines of the kernel proper to provide additional system call interception hooks not included in the LSM interface, and a new system call that allows applications to interact with Thoth. A small application library consisting of 840 LoC exports the API calls shown in Table 2 based on this system call.

LSM module The Thoth LSM module comprises approximately 3500 LoC and intercepts I/O related system calls including `open`, `close`, `read`, `write`, `socket`, `mknod`, `mmap`, etc. Intercepted system calls are redirected to the reference monitor for taint tracking and validation. The module includes conduit interceptors for files, named pipes and sockets, as well as interceptors for client connections to a memcached key-value store [12].

Thoth reference monitor Thoth’s reference monitor is implemented as a trusted, privileged userspace process. It implements the policy enforcement logic and maintains the process taint, session state and transaction state in DRAM. The monitor accesses the persistent Thoth metadata store, which includes per-conduit metadata (conduit pathname, conduit id, a pointer to the policy in effect in the policy store, and for each persistent file conduit, its current size), the transaction log, and the global policy store. The metadata and transaction log are stored in NVRAM. A write-through DRAM cache holds recently accessed metadata and policies.

The monitor is multi-threaded so it can exploit multicore parallelism. Each worker thread invokes the Thoth

Function	Description
<i>confine ()</i>	Transition calling process from UNCONFINED to CONFINED state.
<i>authenticate (key)</i>	Authenticate process with the private key <i>key</i> to satisfy identity-based policies.
<i>add_policy (p)</i>	Store a policy <i>p</i> in Thoth metadata and return an id <i>p_id</i> for it.
<i>set_tx_flags (c_id, flags)</i>	Set flags <i>flags</i> (type and partial evaluation hints) for a transaction on conduit <i>c_id</i> .
<i>open_tx (c_id)</i>	Open a transaction on conduit <i>c_id</i> and return a file handle.
<i>close_tx (fd)</i>	Close a transaction <i>fd</i> . Return 0 if successful, or error code of a policy check fails.
<i>set_policy (fd, p_id)</i>	Attach policy id <i>p_id</i> to the conduit running transaction <i>fd</i> . Passing (-1) for <i>p_id</i> sets the null policy. The new policy is applied only after <i>fd</i> is successfully closed. The declassification condition of the conduit's existing policy determines whether the policy change or removal is allowed.
<i>get_policy (c_id, buf)</i>	Retrieve the policy attached to conduit <i>c_id</i> into buffer <i>buf</i> .
<i>cache (fd, off, len)</i>	Cache content (for policy evaluation) from file handle <i>fd</i> from offset <i>off</i> with length <i>len</i> .

Table 2: Thoth API calls

system call and normally blocks in the LSM module waiting for work. When an application issues a system call that requires an action by the reference monitor, a worker thread is unblocked and returns to the reference monitor with appropriate parameters; when the work is done, the thread invokes the system call again with the appropriate results causing the original application call to either be resumed or terminated. As an optimization, the LSM seeks to amortize the cost of IPC by buffering and dispatching multiple asynchronous requests to a worker thread whenever possible. The reference monitor was implemented in 19,000 LoC of C, not counting the OpenSSL library used for secure sessions and crypto.

Limitations Memory-mapped files are currently supported read-only. Interception is not yet implemented for all I/O-related system calls. None of these missing features are used by our prototype data retrieval system.

5 Policy-compliant data retrieval

We use Thoth for policy compliance in a data retrieval system built around a distributed Apache Lucene search engine. While Apache Lucene's architecture is not appropriate for large, public search engines like Google or Bing, it is frequently used in smaller, domain-specific data retrieval systems.

5.1 Baseline configuration

Lucene Apache Lucene is an open-source search engine written in Java [2]. It consists of an indexer and a search component. The sequential indexer is a single process that scans a corpus of documents and produces a set of index files. The search component consists of a multi-threaded process that executes search queries in parallel and produces a set of corpus file names relevant to a given search query. The size of the Apache Lucene codebase is about 300,000 LoC.

Lucene can be configured with replicated search processes to scale its throughput. Here, multiple nodes run a copy of the search component, each with the full in-

dex. A search query can be processed by any machine. Lucene can also be sharded to scale with respect to the corpus size. In this case, the corpus is partitioned, each partition is indexed individually, and multiple nodes run a copy of the search component, each with one partition index. A search query is sent to all search components, and the results combined. Replication and sharding can be combined in the obvious way.

Front-end processes A simple front-end process accepts user requests from a remote client and forwards search queries to one or more search process(es) via a pipe. The search process(es) may forward the query to other search processes with disjoint shards. When the front-end receives the search results (a list of document file names), it produces a HTML page with a URL and a content snippet from each of the result documents, and returns the page to the Web client. When the client clicks on one of the URLs, the front-end serves the content.

A second, simple account manager front-end process accepts connections from clients for the purpose of creating accounts, managing personal profiles and policies. Clients choose from a set of policy templates for documents they have contributed to the corpus, and for their personal profile information and activity history.

Search personalization and advertising To include typical features of a data retrieval system, we added personalized search and targeted advertising components. A memcached daemon runs on each search node to provide a distributed key-value store for per-user information, including a suffix of the search and click histories, profile information, and the public key. The front-end process appends a user's search queries and clicks to the histories. It uses the profile information to rewrite search queries, re-order search results, and select ads for inclusion in the results page.

An aggregator process periodically analyses a user's search and click history, and updates the personal profile information accordingly. We are not concerned with the details of user profiling, personalized search, or ad

targeting. It suffices for our purposes to capture the appropriate data flows.

5.2 Controlling data flow with Thoth

With Thoth, the front-end, search, indexing, and aggregation tasks execute as `CONFINED` processes, and the account manager executes as an `UNCONFINED` process. Relative to the baseline system, we made minimal modifications, mostly to set an appropriate policy on output conduits. The modifications to Apache Lucene amounted to less than 20 lines of Java code and 30 lines of C code in a JNI library. These modifications set policies on internal conduits and, like the rest of Lucene, are not trusted. Finding the appropriate points to modify was relatively easy because Lucene’s codebase has separate functions through which all I/O is channelled. For applications without this modularity, a dynamically-linked library can be used that overrides libc’s I/O functions and adds appropriate policies.

Unlike in the baseline, the front-end process must be restarted after each user session, to drop its taint. We implement this by exec-ing the process when a new user session starts.

Ingress/egress policies Recall that the ingress and egress policies determine which data flows are allowed and reflect the policies of users, data sources, and provider. In our system, the network connection between the client and the front-end is both an ingress and an egress conduit. The document files in the corpus and the key-value tuples that contain a user’s personal information are ingress conduits. Policies are associated with all ingress and egress conduits as described below. The primary difficulty here is to determine appropriate policies, a task that is required in any compliant system. Specifying the policies in Thoth’s policy language is straightforward.

Account manager flow When Alice creates an account, credentials are exchanged for subsequent mutual authentication, and stored in the key-value store, along with any personal profile information Alice provides.

Alice can choose policies for her profile and history information, as well as any contributed content, typically from a set of policy templates written by the provider’s compliance team. The declassification rule of each policy implicitly controls who can subsequently change the policy; normally, Alice would choose a policy that allows only her to make such a change. Alice may also edit her friend lists or other access control lists stored in the key-value store, which may be referenced by her policies.

Next, we explain the main data flows through the system. For lack of space, we cannot detail all policies on internal conduits, but we highlight the key steps.

Indexing flow Periodically, the indexer is invoked to regenerate the index partitions. A correct indexer only processes documents with the `ONLY_CND_IDS` (or `ONLY_CND_IDS+`) declassification clause, which is transferred to the index files. Note that the index may contain arbitrary data and can be read by any `CONFINED` process; however, an eventual declassification to an `UNCONFINED` process is only possible for a list of conduit ids.

Profile aggregation flow A profile aggregation task periodically executes in the background, to scan the suffix of a user’s query and click history and update the user’s profile vector. A correct aggregator only analyzes user history data that has the `ONLY_CND_IDS` (or `ONLY_CND_IDS+`) declassification clause, which is transferred to the profile vectors.

Search flow Finally, we describe the sequence of steps when Alice performs a search query. The search front-end authenticates itself to Alice using the credentials stored in the key-value store. A successful authentication assures Alice that (i) she is talking to the front-end, and (ii) the front-end process is tainted with the policy of Alice’s credentials (only Alice can read, else declassify into a list of conduit ids) before Alice sends her search query. Next, Alice authenticates herself to the Thoth reference monitor via the search front-end, which proves to Thoth that the front-end process speaks for Alice.

The front-end now sends Alice’s query to one or more search process(es) and adds it to her search history. The search results are declassified as a list of conduit ids, and therefore do not add new taint to the front-end. While producing the HTML results page, the front-end reads a snippet from each result document using Alice’s credentials. Each document has a censorship policy, which checks that the document’s conduit ID is not blacklisted in the client’s region. These policies differ in the conduit IDs and so, in principle, the taint set on the front-end could become very large. To prevent this, we use partial evaluation (Section 3): *Before* a document’s policy is added to the front-end’s taint, we check that the document is not blacklisted. This way, the front-end’s taint increases by a *single predicate* (which verifies Alice’s IP address) when it reads the first document and does not increase when it reads subsequent documents.

Finally, the front-end sends the results page to the client. For this, it must satisfy the egress conduit policy, which verifies Alice’s identity and her IP address.

Result caching High-performance retrieval systems cache search results and content snippets for reuse in similar queries. Although we have not implemented such caching, it can be supported by Thoth. Intermediate results can be cached at various points in the data flow, usually before their policies have been specialized (through

partial evaluation) for a particular client or jurisdiction.

Summary Assuming that the account manager correctly installs ingress and egress policies, Thoth ensures that Alice’s documents, history and profile are used according to her wishes and that the provider’s censorship and MAL policies are enforced, despite any bugs in the indexer, the front-end or the profile aggregator. Thoth’s use in a data retrieval system highlights two different ways of preventing process overtainting. The front-end process is *user-specific*—it acts on behalf of one client. Consequently, the front-end must be re-executed at the end of a user session session to discard its taint. In contrast, the indexer is an *aggregator* process that is designed to combine documents with conflicting policies into a single index. To make its output (the index) usable downstream, the provider installs a typed declassification clause (ONLY_CND_IDS or ONLY_CND_IDS+) on all documents. Due to the declassification clause, there is no need to re-exec the search process.

6 Evaluation

In this section, we present results of an experimental evaluation of our Thoth prototype.

All experiments were performed on Dell R410 servers, each with 2x Intel Xeon X5650 2.66 GHz 6 hyper-threaded core CPUs, 48GB main memory, running OpenSuse Linux 12.1 (kernel version 3.13.1, x86-64). The servers are connected to Cisco Nexus 7018 switches with 1Gbit Ethernet links. Each server has a 1TB Seagate ST31000424SS disk formatted under ext4, which contains the OS installation and a 258GB static snapshot of English language Wikipedia articles from 2008 [43].

We allocate a 2GB memory segment on /dev/shm to simulate NVRAM used by Thoth to store its metadata and transaction log. NVRAM is readily available and commonly used to store frequently updated, fixed-sized persistent data structures like transaction logs.

In the following experiments, we compare a system where each OS kernel is configured with the Thoth LSM kernel module and reference monitor against an otherwise identical baseline system with unmodified Linux 3.13.1 kernels.

6.1 Thoth-based data retrieval system

We study the total Thoth overheads in the prototype retrieval system described in Section 4.

Indexing First, we measure the overhead of the search engine’s index computation. We run the Lucene indexer over a) the entire 258GB snapshot of the English Wikipedia, and b) a 5GB part of the snapshot. The sizes of the resulting indices are 54GB and 959MB, respectively. Table 3 shows the average indexing time and standard deviation across 3 runs. In both cases, Thoth’s runtime overhead is below 1%.

	Dataset 258GB		Dataset 5GB	
	Avg. (mins)	σ	Avg. (mins)	σ
Linux	1956.1	30	27.8	0.06
Thoth	1968.6	24	28.0	0.11
Overhead	0.65%		0.7%	

Table 3: Indexing runtime overhead

Even in a sharded configuration, Lucene relies on a sequential indexer, which can become a bottleneck when a corpus is large and dynamic. Larger search engines may rely on parallel map/reduce jobs to produce their index. As a proof of concept, we built a Hadoop-based indexer using Thoth, although we don’t use it in the following evaluation because it does not support all the features of the Lucene indexer. All mappers and reducers run as confined tasks, and receive the same taint as the original, sequential indexer.

Search throughput Next, we measure the overhead of Thoth on the query latency and throughput. To ensure load balance, we partitioned the index into two shards of 22GB and 33GB, chosen to achieve approximately equal query throughput. We use two configurations:

2Servers: 2 server machines execute a Lucene instance with different index shards. **4Servers**: Here, we use two replicated Lucene instances in each shard to scale the throughput. The front-end forwards each search request to one of the two Lucene instances in each shard and merges the results.

We drive the experiment with the following workload. We simulate a population of 40,000 users, where each user is assigned a friend list consisting of 12 randomly chosen other users, subject to the constraint that the friendship relationship is symmetric. Each item in the corpus is assigned either a private, public, or friends-only policy in the proportion 30/50/20%, respectively. A total of 1.0% of the dataset is censored in some region. All simulated clients are in a region that blacklists 2250 random items.

We use query strings based on the popularity of Wikipedia page accesses during one hour on April 1, 2012 [42]. Specifically, we search for the titles of the top 20K visited articles and assign each of the queries randomly to one of the users. 24 simulated active users connect to each server machine, maintain their sessions throughout the experiment, and issue 48 (**2Servers**) and 96 (**4Servers**) queries concurrently to saturate the system. In addition, a simulated “employee” sporadically issues a read access to protected user files for a total of 200 MAL accesses.

During each query, the front-end looks up the user profile and updates the user’s search history in the key-value store. To maximize the performance of the baseline and fully expose Thoth’s overheads, the index shard and

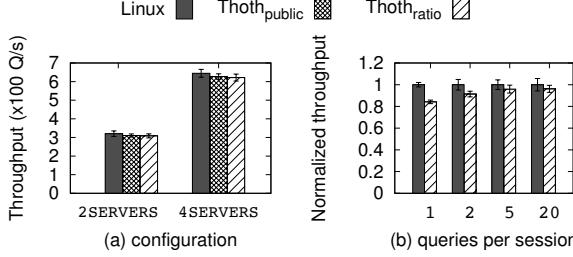


Figure 4: Search throughput

parts of the corpus relevant to our query stream are pre-loaded into the servers’ main memory caches, resulting in a CPU-bound workload.

Figure 4 (a) shows the average throughput over 10 runs of 20K queries each, for the baseline (Linux) and Thoth under **2SERVERS** and **4SERVERS**. The error bars indicate the standard deviation over the 10 runs. We used two Thoth configurations, **Thoth_{public}** and **Thoth_{ratio}**. In **Thoth_{public}**, the policies permit all accesses. This configuration helps to isolate the overhead of Thoth’s I/O interposition and reference monitor invocation. In **Thoth_{ratio}**, input files are private to a user, public, or accessible to friends-only in the ratio 30:50:20. All files allow employee access under MAL, enforce region-based censorship, and have the declassification condition with **ONLY_CONDUIT_IDS+**.

The query throughput scales approximately linearly from **2SERVERS** (320 Q/s) to **4SERVERS** (644 Q/s), as expected. Thoth with all policies enforced (**Thoth_{ratio}**) has an overhead of 3.63% (308 Q/s) in **2SERVERS** and 3.55% in **4SERVERS** (621 Q/s). We note that the throughput achieved with **Thoth_{public}** (310 Q/s and 627 Q/s, respectively) is only slightly higher than **Thoth_{ratio}**’s. This suggests that Thoth’s overhead is dominated by costs like I/O interception, Thoth API calls, and metadata operations, which are unrelated to policy complexity.

To test whether overheads can be reduced further, we also implemented a rudimentary reference monitor in the kernel, which does not support session management and policy interpretation (which require libraries that are unavailable in the Linux kernel). This reduced in-kernel monitor suffices to execute **Thoth_{public}**. Moving the reference monitor to the kernel reduced the overhead of **Thoth_{public}** from 3% to under 1%, which suggests that overheads can be further reduced by moving the reference monitor to the kernel and, hence, eliminating the cost of IPC between the LSM and the reference monitor.

With Thoth, the front-end is re-exec’ed at the end of every user session to shed the front-end’s taint. The relative overhead of doing so reduces with session length. Figure 4 (b) shows the average throughput normalized to the Linux baseline for session lengths of 1, 2, 5 and

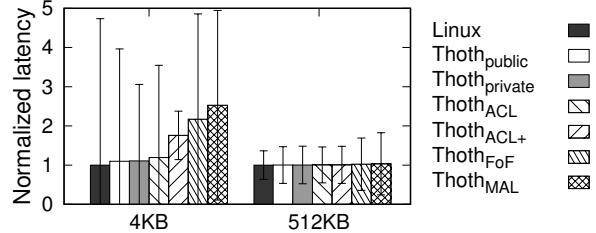


Figure 5: Read latency, normalized to Linux’s

20 queries in **2SERVERS**. Due to the per-session front-end exec, Thoth’s overhead is higher for small sessions (15.8% for a single query); however, the overhead diminishes quickly to 8.6% for 2 queries per session, and the throughput is within a standard deviation of the maximum for 5 or more queries per session in all configurations, including **4SERVERS**.

Search latency Next, we measure the overhead on query latency. Table 4 shows the average query latency across 5 runs of 10K queries in **2SERVERS**. The results in **4SERVERS** are similar. In all cases, Thoth adds less than 6.7ms to the baseline latency.

	Avg. (ms)	σ	Overhead
Linux	47.09	0.43	-
Thoth _{public}	51.60	0.29	9.6%
Thoth _{ratio}	53.78	0.20	14.2%

Table 4: Query search latency (ms)

6.2 Microbenchmarks

Next, we perform a set of microbenchmarks to isolate Thoth’s overheads on different policies. We measure the latency of opening, reading sequentially, and closing 10K files in the baseline and with Thoth under different policies associated with the files. The files were previously written to disk sequentially to ensure fast sequential read performance for the baseline and therefore fully expose the overheads.

In the Thoth experiments, accesses are performed by an UNCONFINED task to force an immediate policy evaluation. The following policies are used. **Thoth_{public}**: files can be read by anyone. **Thoth_{private}**: access is restricted to a specific user. **Thoth_{ACL}**: access to friends only (all users have the same friend list). **Thoth_{ACL+}**: access to friends only (each user has a different friend list). **Thoth_{FoF}**: access to friends of friends (each user has a different friend list). All friend lists used in the microbenchmark have 100 entries. **Thoth_{MAL}**: each file has a MAL policy, where each read requires an entry in a log with an append-only integrity policy.

Figure 5 shows the average time for reading a file of sizes 4K and 512K, normalized to the baseline Linux latency (0.145ms and 3.6ms, respectively); the error bars indicate the standard deviation among the 10K file reads. We see that Thoth’s overheads increase with the complexity of the policy, in the order listed above. For the 4KB files, the overheads range from 10.6% for **Thoth_{public}** and **Thoth_{private}** to 152.7% for **Thoth_{MAL}**. The same trend holds for larger files, but the overhead range diminishes to 0.6%–23% for 96KB files (not shown in the figure) and 0.34%–3.3% for 512KB files.

We also experimented with friend list sizes of 12 and 50 entries for **Thoth_{ACL}**, **Thoth_{ACL+}** and **Thoth_{FoF}**; the resulting latency was within 2.4% of the corresponding 100-entry friend list latency. This is consistent with the known complexity of the friend lookup, which is logarithmic in the list size.

We also looked at the breakdown of Thoth latency overheads. With **Thoth_{ACL}** and 4KB files, Thoth’s overhead for file read is on average 28 μ s, which are spent intercepting the system call and maintaining the session state. Interpreting the policy and checking the friend lists takes 6 μ s, but this time is completely overlapped with the disk read.

Write transaction latency We performed similar microbenchmarks for write transactions. In general, Thoth’s write transactions have low overhead since its transaction log is stored in (simulated) NVRAM. As in the case of read latency, the overhead depends on the granularity of writes and the complexity of the policy being enforced. Under the index policy, the overhead ranges from 0.25% for creating large files to 2.53x in the case of small files. The baseline Linux is very fast at creating small files that are written to disk asynchronously, while Thoth has to synchronously update its policy store when a new file is created. The overhead is 5.8x and 8.6x in the case of a write of 10 conduit ids to a file under the **ONLY_CND_IDS** and **ONLY_CND_IDS+** policies, respectively. This high overhead is due to checking that each conduit id being written exists (and is written into a file with a stricter policy in the case of **ONLY_CND_IDS+**). However, this overhead amounts to only a small percentage of the overall search query processing, as is evident from Table 4.

6.3 Fault-injection tests

To double-check Thoth’s ability to stop unwanted data leaks, we injected several types of faults in different stages of the search pipeline.

Faulty Lucene indexer We reproduced a known Lucene bug [5] that associates documents with wrong attributes during index creation. This bug is security-relevant because, in the absence of another mechanism, attributes can be used for labeling data with their owners.

In our experiment Thoth successfully stopped the flow in all cases where the search results contained a conduit whose policy disallowed access to the client.

We also intentionally misconfigured the indexer to index the users’ query and click histories, which should not show up in search results. Thoth prevented the indexer from writing the index after it had read either the query or the click history.

Faulty Lucene search We reproduced a number of known Lucene bugs that produce incorrect search results. Such bugs may produce Alice’s private documents in Bob’s search. The bugs include incorrect parsing of special characters [7], incorrect tokenization [9], confusing uppercase and lowercase letters [10], using an incorrect logic for query expansion [4, 3], applying incorrect keyword filters [8], and premature search termination [6]. We confirmed that all policy violations resulting from these bugs faults were blocked by Thoth.

To check the declassification condition **ONLY_CND_IDS+**, we modified the search process to (incorrectly) output text from the index in place of conduit ids. Thoth prevented the search process from producing such output.

Faulty front-end We issued accesses to a private file protected by the **MAL** policy without adding appropriate log entries. Thoth prevented the front-end process from extricating data to the caller. We performed similar tests for the region-based censorship policy with similar results.

7 Related work

Search engine policy compliance Grok [36] is a privacy compliance tool for the Bing search engine. Grok and Thoth differ in techniques, expressiveness and target policies. Grok uses heuristics and selective manual verification by developers to assign *attributes* — abstract labels that represent intended confidentiality — to processes and data stores. Grok policies, written in a language called Legalese, specify allowed data flows on attributes. Attributes and policies apply at the granularity of fields (types), not individual users or data items, so Legalese cannot express the private, friends only and friends of friends policies from Section 2. (This restriction applies broadly to most static analysis-based policy enforcement techniques.) Legalese also does not support content-dependent policies and cannot express the mandatory access logging, censorship and typed declassification policies from Section 2. Grok enforces policies with a fast static analysis on computations written in languages like Hive, Dremel, and Scope. Grok imposes no runtime overhead. Thoth uses kernel-level interception and is language-independent, but has a small runtime overhead. Grok-assigned attributes may be incorrect, so Grok may have false negatives. In contrast,

Thoth enforces all conduit policies without false negatives.

Cloud policy compliance Maniatis et al. [31] outline a vision, architecture and challenges for data protection in the Cloud using secure data capsules. Thoth can be viewed as a realization of that vision in the context of a data retrieval system, and contributes the design of a policy language, enforcement mechanism, and experimental evaluation. Secure Data Preservers (SDaPs) [27] are software components that mediate access to data according to a user-provided policy. Unlike Thoth, SDaPs are suitable only for web services that interact with user data through simple, narrow interfaces, and do not require direct access to users' raw data. LoNet [26] enforces data-use policies at the VM-level. Unlike Thoth, declassification requires trusted application code and interception is limited to file I/O using FUSE, which results in very high overhead.

Information flow control (IFC) Numerous systems restrict a program's data flow to enforce security policies, either in the programming language (Jif [34]), in the language runtime (Resin [46], Nemesis [19]), in language libraries (Hails [25]), using software fault isolation (duPro [35]), in the OS kernel (e.g., Asbestos [22], HiStar [47], Flume [28], Silverline [33]), or in a hypervisor (Neon [48]). Thoth differs from these systems in a number of ways. Unlike language-based IFC, Thoth applications can be written in any language.

Architecturally, Thoth is close to Flume. Both isolate processes using a Linux security extension and a user-space reference monitor, both enforce policies on conduits and both distinguish between `CONFINED` and `UNCONFINED` processes in similar ways. However, like all other kernel-level solutions for IFC (Asbestos, HiStar, Silverline), Flume uses abstract labels as taints. In contrast, Thoth uses declarative policies as taints. This results in two fundamental differences. First, Flume relies on trusted application components to map system access policies to abstract labels and for all declassification. In contrast, in Thoth, the reference monitor enforces all access conditions (specified in the `read` and `update` rules) and all declassification conditions (specified in the `declassify` clauses). Application components are trusted only to install correct policies on ingress and egress nodes. Second, Thoth policies describe the policy configuration completely. In Flume, the policy configuration is implicit in the `code` of the trusted components that declassify and endorse data, and map access policies to labels (although mapping can be automated to some extent [21]).

Resin [46] enforces programmer-provided policies on PHP and Python web applications. Unlike Thoth's declarative policies, Resin's policies are specified as

PHP/Python functions. Resin tracks flows at object granularity. Thoth tracks flows at process granularity, which matches the pipelined structure of data retrieval systems and reduces overhead significantly. Hails [25] is a Haskell-based web development framework with statically-enforced IFC. Thoth offers IFC in the kernel, and is independent of any language, runtime, or framework used for developing applications. COWL [39] confines JavaScript browser contexts using labels and IFC. Thoth addresses the complementary problem of controlling data flows on the server side. Both Hails and COWL use DC-labels [38] as policies. DC-labels cannot express content-dependent policies like our censorship, mandatory access logging and ONLY_CND_IDS policies.

Declarative policies Thoth's policy language is based on Datalog and linear temporal logic (LTL). Datalog and LTL are well-studied foundations for policy languages (see [30, 18, 20] and [15, 16, 23], respectively), known for their clarity, conciseness, and high-level of abstraction. The primary innovation in Thoth's policy language is its two-layered structure, where the first layer specifies access policies and the second layer specifies declassification policies. Some operating systems (Nexus and Taos [37, 45]), file systems (PFS and PCFS [41, 24]), and at least one cyber-physical system (Grey [17]) and one storage system (Guardat [40]) enforce access policies expressed in Datalog-like languages. Thoth can enforce similar policies but, additionally, Thoth tracks flows and can enforce declassification policies that these systems cannot enforce. Like Guardat, but unlike the other systems listed above, Thoth's policy language supports data-dependent policies. The design of Thoth's reference monitor is inspired by Guardat's monitor. However, Thoth's monitor tracks data flows, supports declassification policies, and intercepts memcached I/O and network communication, all of which Guardat's monitor does not do.

8 Ongoing work

In this section, we briefly describe ongoing work related to Thoth.

Lightweight isolation Information flow control requires the isolation of computations that handle different users' private data. In general-purpose operating systems, this means that separate processes must be used to handle user sessions. Thoth, for instance, requires that front-end processes be exec'ed for each new session. We are working on an operating system primitive that provides isolation among different user sessions within the same process with low cost.

Database-backed retrieval systems Thoth includes conduit interceptors for files, named pipes, network connections and a key-value store (memcached). In current

work, we are building a system to ensure compliance of SQL database queries with declarative policies associated with the database schema. The system can be used as a conduit interceptor, thus extending Thoth’s protection to database-backed data retrieval systems.

Policy testing Assigning policies to internal conduits in Thoth, and making sure that they permit all data flows compliant with the ingress and egress policies, can be a tedious task in a large system. In current work, we are developing a tool that generates internal conduit policies semi-automatically using a system’s dataflow graph and the ingress/egress policies as inputs. Moreover, the tool performs systematic testing to ensure all compliant dataflows are allowed, and helps the policy developer generate appropriate declassification policies as needed.

9 Conclusion

Efficient policy compliance in data retrieval systems is a challenging problem. Thoth is a kernel-level policy compliance layer to address this problem. The provider has the option to associate a declarative policy with each data source and sink. The policy specifies confidentiality and integrity requirements and may reflect the data owner’s privacy preferences, the provider’s own data-use policy, and legal requirements. Thoth enforces these policies by tracking and controlling data flows across tasks through kernel I/O interception. It prevents data leaks and corruption due to bugs and misconfigurations in application components (including misconfigurations in policies on internal conduits), as well as actions by unprivileged operators.

Our technical contributions include a declarative policy language that specifies both access (read/write) policies and how those access policies may change. The latter can be used to represent declassification policies. Additionally, the language supports content-dependent policies. Thoth uses policy sets as taint, which eliminates the need to trust application processes with access checks at the system boundary and with declassification. Our Linux-based prototype shows that Thoth can be deployed with low overhead in data retrieval systems. Among other things, this demonstrates the usefulness and viability of coarse-grained taint tracking as a basis for policy enforcement.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful feedback. This research was supported in part by the European Research Council (ERC Synergy imPACT 610150) and the German Research Foundation (DFG CRC 1223).

References

- [1] Adobe data breach more extensive than previously disclosed. <http://www.reuters.com/article/2013/10/29/us-adobe-cyberattack-idUSBRE99S1DJ20131029>.
- [2] Apache Lucene. <http://lucene.apache.org>.
- [3] Apache Lucene bug report 1300. <https://issues.apache.org/jira/browse/LUCENE-1300>.
- [4] Apache Lucene bug report 2756. <https://issues.apache.org/jira/browse/LUCENE-2756>.
- [5] Apache Lucene bug report 3575. <https://issues.apache.org/jira/browse/LUCENE-3575>.
- [6] Apache Lucene bug report 4511. <https://issues.apache.org/jira/browse/LUCENE-4511>.
- [7] Apache Lucene bug report 49. <https://issues.apache.org/jira/browse/LUCENE-49>.
- [8] Apache Lucene bug report 6503. <https://issues.apache.org/jira/browse/LUCENE-6503>.
- [9] Apache Lucene bug report 6595. <https://issues.apache.org/jira/browse/LUCENE-6595>.
- [10] Apache Lucene bug report 6832. <https://issues.apache.org/jira/browse/LUCENE-6832>.
- [11] DataLossDB: Open Security Foundation. <http://datalossdb.org>.
- [12] Memcached. <http://memcached.org/>.
- [13] Privacy Rights Clearinghouse. <http://privacyrights.org>.
- [14] Target breach worse than thought, states launch joint probe. <http://www.reuters.com/article/2014/01/10/us-target-breach-idUSBREA090L120140110>.
- [15] Adam Barth, John C. Mitchell, Anupam Datta, and Sharada Sundaram. Privacy and utility in business processes. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, 2007.
- [16] David A. Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV)*, 2010.
- [17] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 26th IEEE Symposium on Security and Privacy (S&P)*, 2005.

- [18] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Design and semantics of a decentralized authorization language. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, 2007.
- [19] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th USENIX Security Symposium*, 2009.
- [20] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (S&P)*, 2002.
- [21] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2008.
- [22] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.
- [23] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: theory, implementation and applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.
- [24] Deepak Garg and Frank Pfennig. A proof-carrying file system. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.
- [25] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [26] Havard D. Johansen, Eleanor Birrell, Robbert van Renesse, Fred B. Schneider, Magnus Stenhaug, and Dag Johansen. Enforcing privacy policies with meta-code. In *Proceedings of the 6th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2015.
- [27] Jayanthkumar Kannan, Petros Maniatis, and Byung-Gon Chun. Secure data preservers for web services. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.
- [28] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.
- [29] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 1998.
- [30] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th Symposium on Practical Aspects of Declarative Languages*, 2003.
- [31] Petros Maniatis, Devdatta Akhawe, Kevin Fall, Elaine Shi, Stephen McCamant, and Dawn Song. Do you know where your data are? secure data capsules for deployable data protection. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2011.
- [32] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
- [33] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. Silverline: Preventing data leaks from compromised web applications. In *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.
- [34] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999.
- [35] Ben Niu and Gang Tan. Efficient user-space information flow control. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013.
- [36] Shayak Sen, Saikat Guha, Anupam Datta, Sriman K. Rajamani, Janice Tsai, and Jeannette M. Wing. Bootstrapping privacy compliance in big data systems. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [37] Alan Shieh, Dan Williams, Emin Gün Sirer, and Fred B Schneider. Nexus: a new operating system for trustworthy computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.
- [38] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications*, 2011.
- [39] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [40] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the 3rd ACM SIGOPS European Conference on Computer*

- Systems (EuroSys), 2015.*
- [41] Kevin Walsh and Fred B. Schneider. Costs of security in the PFS file system. Technical report, Computing and Information Science, Cornell University, 2012.
 - [42] Wikimedia Foundation. Image Dump. <http://archive.org/details/wikimedia-image-dump-2005-11>.
 - [43] Wikimedia Foundation. Static HTML dump. <http://dumps.wikimedia.org/>.
 - [44] Wikipedia. Data breach: Major incidents. http://en.wikipedia.org/wiki/Data_breach#Major_incidents.
 - [45] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(1), 1994.
 - [46] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.
 - [47] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
 - [48] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Neon: System support for derived data management. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2010.

A Thoth policies for data flows in a search engine

In this Appendix we provide details of the policies used in our Thoth-compliant search engine. All policies are represented in the **read**, **update** and **declassify** rules on source conduits (documents that the search engine indexes, the user profile, etc.). We describe these rules incrementally: We start from a set of base rules, which we refine to include more policies.

Base rules Our base rules allow anyone to read, update or destroy the source conduit they are attached to.

```
read :- TRUE
update :- TRUE
destroy :- TRUE
declassify :- isAsRestrictive(read, this.read)
until FALSE
```

The **read**, **update** and **destroy** rules have condition *TRUE*, which always holds, so these rules do not re-

strict access at all. The **declassify** rule insists that the **read** rule on any conduit containing data derived from the source conduit be at least as restrictive as the **read** rule above, which will always be the case (because the **read** rule above is the most permissive read rule possible). This base policy is pointless in itself, but it serves as the starting point for the remaining policies.

A.1 Client policies

First, we describe policies to represent client privacy preferences.

Private data policy A user Alice may wish that her private files (e.g., her e-mails) be accessible only to her. This can be enforced by requiring that accesses to Alice’s private files happen in the context of a session authenticated with Alice’s key. Technically, this is accomplished by replacing the conditions in the base **read**, **update** and **destroy** rules as shown below and attaching the resulting rules to Alice’s private files. The predicate *sKeyls(k)* means that the current session is authenticated using the public key *k*.

```
read :- sKeyls(kAlice)
update :- sKeyls(kAlice)
destroy :- sKeyls(kAlice)
```

The **declassify** rule remains unchanged. It ensures that any conduit containing data derived from Alice’s private files is subject to a **read** rule that is at least as restrictive as the revised **read** rule above. Hence, no such conduit can be read by anyone other than Alice.

Friends only policy Alice might want that her blog and online social network profile be readable by her friends. To do this, she could add a disjunctive (“or”-separated) clause in the **read** rule requiring that **read** accesses happen in the context of a session authenticated with a key *k_X* of one of Alice’s friends. Alice’s friends are assumed to be listed in the file *Alice.acl*, which contains an entry of the form *isFriend(k_X, X_{ACL})* for each public key *k_X* that belongs to a friend of Alice. The *isFriend* entry also states the file *X_{ACL}* which lists the friends of the key *k_X*’s owner. Note that the *isFriend* entry format presented in the paper was slightly simplified for readability.

```
read :- sKeyls(kAlice) ∨
[sKeyls(kX) ∧ (“Alice.acl”, off) says isFriend(kX, XACL)]
```

The *says* predicate ((“Alice.acl”, *off*) *says* *isFriend(k_X, X_{ACL})*) checks that *k_X* exists in the list of Alice’s friends (file “Alice.acl”) at some offset *off*.

Friends of friends policy To additionally allow read access to friends of friends, the policy would require read accesses to happen in the context of an authenticated session whose key is present in the friend list of any of Alice’s friends.

```

read :- sKeyls( $k_{Alice}$ ) ∨
[sKeyls( $k_X$ ) ∧ (“Alice.acl”,  $off$ ) says isFriend( $k_X, X_{ACL}$ )] ∨
[sKeyls( $k_Y$ ) ∧ (“Alice.acl”,  $off_1$ ) says isFriend( $k_X, X_{ACL}$ )
 ∧ ( $X_{ACL}, off_2$ ) says isFriend( $k_Y, Y_{ACL}$ )]

```

The predicate $((‘Alice.acl’, off_1) \text{ says } \text{isFriend}(k_X, X_{ACL}))$ checks that k_X exists in the list of Alice’s friends (file “Alice.acl”) at some offset off_1 . It also binds the variable X_{ACL} to the friend list of the key k_X ’s owner. Next, the predicate $((X_{ACL}, off_2) \text{ says } \text{isFriend}(k_Y, Y_{ACL}))$ checks that the public key that authenticated the session k_Y exists in the list of friends for the k_X ’s owner at some offset off_2 .

A.2 Provider policies

Next, we describe two policies that a provider may wish to impose, possibly to comply with legal requirements.

Mandatory Access Logging (MAL) The MAL policy allows an authorized employee of the provider read access to a source conduit F if the access is logged. The log entry must have been previously written to the file $k.log$, where k is the public key of the employee. The log entry must mention the employee’s key, the ID of the accessed conduit and the time at which the conduit is accessed with a tolerance of 60 seconds. To enforce these requirements, a new disjunctive condition is added to the last **read** rule above. The \dots in the rule below abbreviate the conditions of the last **read** rule above.

```

read :- ... ∨
sKeyls( $k$ ) ∧ cldls( $F$ ) ∧
(“auth_employees”,  $off$ ) says isEmployee( $k$ ) ∧
( $LOG_k = \text{concat}(k, “.log”)$ ) ∧
( $LOG_k, off_1$ ) says readLog( $k, F, T$ ) ∧ timels( $curT$ ) ∧
gt( $curT, T$ ) ∧ sub( $diff, curT, T$ ) ∧ lt( $diff, 60$ )

```

The predicate $sKeyls(k)$ binds the public key that authenticated the session (i.e., the public key of the employee) to the variable k , and $cldls(F)$ binds the name of source conduit to F . Next, the predicate $((“auth_employees”, off) \text{ says } \text{isEmployee}(k))$ checks that k exists in the list of authorized employees (file “auth_employees”) at some offset off , to verify that the source conduit’s reader is really an employee. Next, LOG_k is bound to the name of the employee’s log file, $k.log$. The predicate $((LOG_k, off_1) \text{ says } \text{readLog}(k, F, T))$ checks that the log file contains an appropriate entry with some time stamp T and the remaining predicates check that the current time, $curT$, satisfies $T \leq curT \leq T + 60s$.

Every log file has a **read** rule that allows only authorized auditors to read the file (the public keys of all authorized auditors are assumed to be listed in the file “auditors”). It also has an **update** rule that allows appends

only, thus ensuring that a log entry cannot be removed or overwritten.

```

read :- sKeyls( $k$ ) ∧ (“auditors”,  $off$ ) says isAuditor( $k$ )
update :- sKeyls( $k$ ) ∧
(“auth_employees”,  $off$ ) says isEmployee( $k$ ) ∧
cCurrLenls( $cLen$ ) ∧ cNewLenls( $nLen$ ) ∧
gt( $nLen, cLen$ ) ∧ (this, 0,  $cLen$ ) hasHash ( $h$ ) ∧
(this, 0,  $cLen$ ) willHaveHash ( $h$ )

```

In the append-only policy (rule **update** above), the predicate $cCurrLenls(cLen)$ binds the current length of the log file to $cLen$ and the predicate $cNewLenls(nLen)$ binds the new length of the log file to $nLen$. Next, the predicate $gt(nLen, cLen)$ ensures that the update only increases the log file’s length. (c, off, len) hasHash (or willHaveHash) is a special mode of using says (or willsay) which allows the policy interpreter to refer to the hash of the conduit c ’s content (or updated content in a write transaction) from offset off with length len . In the **update** rule, hasHash and willHaveHash are used to verify that the existing file content is not modified during an update by checking that the hashes of the file from offset 0 to $cLen$, originally and after the prospective update, are equal.

A more efficient implementation of the append-only policy could rely on a specialized predicate $\text{unmodified}(off, len)$, which checks that the conduit contents from offset off with length len were not modified. The **update** rule could then be simplified to:

```

update :- sKeyls( $k$ ) ∧
(“auth_employees”,  $off$ ) says isEmployee( $k$ ) ∧
cCurrLenls( $cLen$ ) ∧ cNewLenls( $nLen$ ) ∧
gt( $nLen, cLen$ ) ∧ unmodified(0,  $cLen$ )

```

Region-based censorship Legal requirements may force the provider to blacklist certain source files in certain regions. Accordingly, the goal of the censorship policy is to ensure that content from a document F can only reach users in regions whose blacklists do not contain F . The policy relies on a mapping from IP addresses to regions and a per-region blacklist file. The blacklist file is maintained in a sorted order to efficiently lookup whether it contains a given document or not.

The censorship policy is expressed by modifying the **declassify** rule of every source conduit $cndID$ as follows:

```

declassify :- isAsRestrictive(read, this.read) until
(CENSOR( $cndID$ ) ∧ isAsRestrictive(read, this.read))

```

The rule says that the **read** rule on any conduit to which $cndID$ flows must be as restrictive as $cndID$ ’s **read** rule *until* a conduit at which the condition $CENSOR(cndID)$ holds is reached. $CENSOR(cndID)$ is a macro defined below. The predicate $slpls(IP)$ checks

that the IP address of the connecting (remote) party is IP and the predicate $\text{IpPrefix}(IP, R)$ means that IP belongs to region R . The blacklist file for region R is $R.\text{BlackList}$. In words, $\text{CENSOR}(\text{cndID})$ means that the remote party's IP belongs to a region R and cndID lies strictly between two consecutive entries in R 's blacklist file (and, hence, cndID does not exist in R 's blacklist file).

```
slpls(IP) ∧ IpPrefix(IP, R) ∧
(FBL = concat(R, ".BlackList")) ∧
(FBL, off1) says isCensored(cnd1) ∧
add(off2, off1, CENSOR_ENTRY_LEN) ∧
(FBL, off2) says isCensored(cnd2) ∧
lt(cnd1, cndID) ∧ lt(cndID, cnd2)
```

A.3 Search engine flows

Indexing flow The indexer reads documents with possibly contradictory policies and, in the absence of a dedicated provision for declassification, the index (and any documents derived from it) cannot be served to any client. To prevent this problem, searchable documents allow typed declassification. The **declassify** rule for each searchable document is modified with a new clause that allows complete declassification into an (internal) conduit whose **update** rule allows the conduit to contain only a list of object ids. The modified **declassify** rule of each source document has the form:

```
declassify :- ... until (... ∨ (clsIntrinsic ∧
isAsRestrictive(update, ONLY_CND_IDS)))
```

The macro `ONLY_CND_IDS` stipulates that only a list of valid conduit ids can be written and it expands to:

```
cCurrLenls(cLen) ∧ cNewLenls(nLen) ∧
each in(this, cLen, nLen) says(cndId)
{cldExists(cndId)}
```

In the macro above, the predicate $\text{cNewLenls}(nLen)$ binds the new length of the output file to $nLen$. The predicate willsay checks that the content update from offset 0 and length $nLen$ is a list of conduit IDs, and the predicate $\text{cldExists}(cndId)$ checks that $cndId$ corresponds to an existing conduit.

So far we have assumed that the conduit ids are not themselves confidential. If the presence or absence of a particular conduit id in the search results may leak sensitive information, then the source declassification policy can be augmented to require that the list of conduit ids is accessible only to a principal who satisfies the confidentiality policies of all listed conduits. Then, the macro `ONLY_CND_IDS` can be re-written to:

```
cCurrLenls(cLen) ∧ cNewLenls(nLen) ∧
each in(this, cLen, nLen) willsay(cndId)
{cldExists(cndId) ∧ hasPol(cndId, P) ∧
isAsRestrictive(read, P.read) ∧
isAsRestrictive(declassify, P.declassify)}
```

Additionally in the macro above, the predicate $\text{hasPol}(cndId, P)$ binds P to the policy of the conduit $cndId$, and the predicate $\text{isAsRestrictive}(\text{read}, P.\text{read})$ requires that the confidentiality of the list of conduit ids is as restrictive as the confidentiality requirements of the source conduit ids themselves.

Profile aggregation flow Since raw user activity logs are typically private, a declassification is required that enables a profile generator to produce a user preferences vector (a vector of fixed length) from the activity logs. However, this preferences vector must further be restricted so that it can be used to produce only a list of conduit ids (the search results). Further, the user might also want to ensure that only activity logs generated in the past 48 hours be used for personalization. This can be achieved by allowing the declassification into the fixed-size vector to happen only within 172800 seconds of the log's creation. Suppose an activity log is created at time t and that the preferences vector has length n . Then, the relevant policy rules on the activity log are the following (note that t and n are constants, not variables).

```
read :- sKeyls(kAlice)
declassify :- [isAsRestrictive(read, this.read) until
isAsRestrictive(update, ONLY_FLOATS(n)) ∧
clsIntrinsic ∧ timels(curT) ∧ gt(curT, t) ∧
sub(diff, curT, t) ∧ lt(diff, 172800)] ∧
[isAsRestrictive(read, this.read) until clsIntrinsic ∧
isAsRestrictive(update, ONLY_CND_IDS)]
```

This policy ensures that the raw user logs can only be transformed into the user preferences vector, which in turn can only be declassified into the search results of the search engine.

The macro `ONLY_FLOATS(n)` stipulates that only a vector of n floats can be written. It expands to:

```
cNewLenls(nLen) ∧
each in(this, 0, nLen) willsay(value)
{vType(value, FLOAT) ∧ (Cnt ++)} ∧
eq(Cnt, n)
```

In the macro above, the predicate $\text{cNewLenls}(nLen)$ binds the new length of the output file to $nLen$. The predicate willsay checks that the content update from offset 0 and length $nLen$ is a list of $values$, and the predicate $\text{vType}(value, FLOAT)$ checks that each $value$ in the list is of type `FLOAT`. The predicate $\text{eq}(cnt, n)$ checks that the update contains n floats.

Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage

Christina Garman

Johns Hopkins University

cgarman@cs.jhu.edu

Matthew Green

Johns Hopkins University

mgreen@cs.jhu.edu

Gabriel Kaptchuk

Johns Hopkins University

gkaptchuk@cs.jhu.edu

Ian Miers

Johns Hopkins University

imiers@cs.jhu.edu

Michael Rushanan

Johns Hopkins University

micharu1@cs.jhu.edu

Abstract

Apple’s iMessage is one of the most widely-deployed end-to-end encrypted messaging protocols. Despite its broad deployment, the encryption protocols used by iMessage have never been subjected to rigorous cryptanalysis. In this paper, we conduct a thorough analysis of iMessage to determine the security of the protocol against a variety of attacks. Our analysis shows that iMessage has significant vulnerabilities that can be exploited by a sophisticated attacker. In particular, we outline a novel chosen ciphertext attack on Huffman compressed data, which allows *retrospective* decryption of some iMessage payloads in less than 2^{18} queries. The practical implication of these attacks is that any party who gains access to iMessage ciphertexts may potentially decrypt them remotely and after the fact. We additionally describe mitigations that will prevent these attacks on the protocol, without breaking backwards compatibility. Apple has deployed our mitigations in the latest iOS and OS X releases.

1 Introduction

The past several years have seen widespread adoption of end-to-end encrypted text messaging protocols. In this work we focus on one of the most popular such protocols: Apple’s iMessage. Introduced in 2011, iMessage is an end-to-end encrypted text messaging system that supports both iOS and OS X devices. While Apple does not provide up-to-date statistics on iMessage usage, in February 2016 an Apple executive noted that the system had a peak transmission rate of more than 200,000 messages per second, across 1 billion deployed devices [12].

The broad adoption of iMessage has been controversial, particularly within the law enforcement and national security communities. In 2013, the U.S. Drug Enforcement Agency deemed iMessage “a challenge for DEA intercept” [22], while in 2015 the U.S. Department of

Justice accused Apple of thwarting an investigation by refusing to turn over iMessage plaintext [11]. iMessage has been at the center of a months-long debate initiated by U.S. and overseas officials over the implementation of “exceptional access” mechanisms in end-to-end encrypted communication systems [7, 26, 33], and some national ISPs have temporarily blocked the protocol [32]. Throughout this controversy, Apple has consistently maintained that iMessage encryption is end-to-end and that even Apple cannot recover the plaintext for messages transmitted through its servers [10].

Given iMessage’s large installed base and the high stakes riding on its confidentiality, one might expect iMessage to have received critical attention from the research community. Surprisingly, there has been very little analysis of the system, in large part due to the fact that Apple has declined to publish the details of iMessage’s encryption protocol. In this paper we aim to remedy this situation. Specifically, we attempt to answer the following question: how secure is Apple iMessage?

Our contributions. In this work we analyze the iMessage protocol and identify several weaknesses that an attacker may use to decrypt iMessages and attachments. While these flaws do not render iMessage completely insecure, some flaws reduce the level of security to that of the TLS encryption used to secure communications between end-user devices and Apple’s servers. This finding is surprising given the protection claims advertised by Apple [10]. Moreover, we determine that the flaws we detect in iMessage may have implications for other aspects of Apple’s ecosystem, as we discuss below.

To perform our analysis, we derived a specification for iMessage by conducting a partial black-box reverse engineering of the protocol as implemented on multiple iOS and OS X devices. Our efforts extend a high-level protocol overview published by Apple [9] and two existing partial reverse-engineering efforts [1, 34]. Armed with a protocol specification, we conducted manual cryptanal-

ysis of the system. Specifically, we tried to determine the system’s resilience to both back-end infrastructure attacks and more restricted attacks that subvert only client-local networks.

Our analysis uncovered several previously unreported vulnerabilities in the iMessage protocol. Most significantly, we identified a *practical* adaptive chosen ciphertext attack on the iMessage encryption mechanism that allows us to retrospectively decrypt certain iMessage payloads and attachments, provided that a single Sender or Recipient device is online. To validate this finding, we implemented a proof of concept exploit against our own test devices and show that the attack can be conducted remotely (and silently) against any party with an online device. This exploit is non-trivial and required us to develop novel exploit techniques, including a new chosen ciphertext attack that operates against ciphertexts containing gzip compressed data. We refer to this technique as a *gzip format oracle* attack, and we believe it may have applications to other encryption protocols. We discuss the details of this attack in §5.

We also demonstrate weaknesses in the device registration and key distribution mechanisms of iMessage. One weakness we exploit has been identified by the reverse engineering efforts in [34], while another is novel. As they are not the main result of this work, we include them in Appendix A for completeness.

Overall, our determination is that while iMessage’s end-to-end encryption protocol is an improvement over systems that use encryption on network traffic only (e.g., Google Hangouts), messages sent through iMessage may not be secure against sophisticated adversaries. Our results show that an attacker who obtains iMessage ciphertexts can, at least for some types of messages, *retrospectively* decrypt traffic. Because Apple stores encrypted, undelivered messages on its servers and retains them for up to 30 days, such messages are vulnerable to any party who can obtain access to this infrastructure, e.g., via court order [11] or by compromising Apple’s globally-distributed server infrastructure [36]. Similarly, an attacker who can intercept TLS using a stolen certificate may be able to intercept iMessages on certain versions of iOS and Mac OS X that do not employ certificate pinning on Apple Push Network Services (APNs) connections.

Given the wide deployment of iMessage, and the attention paid to iMessage by national governments, these threats do not seem unrealistic. Fortunately, the vulnerabilities we discovered in iMessage are relatively straightforward to repair. In the final section of this paper, we offer a set of mitigations that will restore strong cryptographic security to the iMessage protocol. Some of these are included in iOS 9.3 and Mac OS X 10.11.4, which shipped in March 2016.

Other uses of the iMessage encryption protocol. While

our work primarily considers the iMessage instant messaging system, we note that the vulnerabilities identified here go beyond iMessage. Apple documentation notes that Apple’s “Handoff” service, which transmits personal data between Apple devices over Bluetooth Low Energy, encrypts messages “in a similar fashion to iMessage” [9]. This raises the possibility that our attacks on iMessage encryption may also affect inter-device communication channels used between Apple devices. Attacks on this channel are particularly concerning because these functions are turned on by default in many new Apple devices. We did not investigate these attack vectors in this work but subsequent discussions with Apple have confirmed that Apple uses the same encryption implementation to secure both iMessage and inter-device communications. Thus, securing these channels is one side effect of the mitigations we propose in §7.

1.1 Responsible disclosure

In November 2015 we delivered a summary of the results in this paper to Apple. Apple acknowledged the vulnerability in §5 and has initiated substantial repairs to the iMessage system. These repairs include: enforcing certificate pinning across all channels used by iMessage,¹ removing compression from the iMessage composition (for attachment messages), and developing a fix based on our proposed “duplicate ciphertext detection” mitigation (see §7). Apple has also made changes to the use of iMessage in inter-device communications such as Hand-off, although the company has declined to share the details with us. The repairs are included in iOS 9.3 and OS X 10.11.4, which shipped in March 2016.

1.2 Attack Model

Our attacks in §5 require the ability to obtain iMessage ciphertexts sent to or received by a client. Because Apple Push Network Services (APNs) uses TLS to transmit encrypted messages to Apple’s back-end servers, exploiting iMessage requires either access to data from Apple’s servers or a forged TLS certificate. We stress that while this is a strong assumption, it is the appropriate threat model for considering end-to-end encrypted protocols.

A more interesting objection to this threat model is the perception that iMessage might be too weak to satisfy it. For example, in 2013 Raynal *et al.* pointed out a simple attack on Apple’s key distribution that enables a TLS MITM attacker to replace the public key of a recipient with an attacker-chosen key [34]. One finding of this work is that as of December 2015 such attacks have been entirely mitigated by Apple through the addition of

¹This feature was added to OS X 10.11 in December, as a result of our notification.

certificate pinning on key server connections (see Appendix A). More fundamentally, however, such attacks are *prospective* – in the sense that they require the attacker to target a particular individual before the individual begins communicating. By contrast, the attacks we describe in this paper are *retrospective*. They can be run against any stored message content, at any point subsequent to communication, provided that one target device remains online. Moreover, unlike previous attacks which require access to the target’s local network, our attacks may be run remotely through Apple’s infrastructure.

2 The iMessage Protocol

To obtain the full iMessage specification, we began with the security overview provided by Apple, as well as detailed previous software reverse-engineering efforts conducted by Raynal [34] and others [1]. While these previous results provide some details of the protocol, they omit key details of the encryption mechanism, as well as the complete key registration and notification mechanisms. We conducted additional black-box reverse engineering efforts to recover these elements. Specifically, we analyzed and modified protocol exchanges to and from several jailbroken and non-jailbroken Apple devices.² In conformity to Apple’s terms of service, we did not perform any software decompilation.

2.1 System overview

iMessage clients. iMessage clients comprise several pieces of software running on end-user devices. On iOS and OS X devices, the primary user-facing component is the Messages application. On OS X computers, this application interacts with at least three daemons: apsd, the daemon responsible for pushing and pulling application traffic over the Apple Push Notification Service (APNs) channel; imagent, a daemon that pulls notifications even if Messages is closed; and identitieservicesd, a daemon which maintains a cache of other users’ keys. iOS devices also contain an apsd daemon, while other daemons handle the task of managing identities.

Apple services. iMessage clients interact with multiple back-end services operated by Apple and its partners. We focus on the two most relevant to our attack. The Apple directory service (IDS, also known as ESS) maintains a mapping between user identities and public keys and is responsible for distributing user public keys on request. iMessage content is transmitted via the Apple Push Notification Service (APNs). Long iMessages and attachments are transmitted by uploading them to the iCloud

²In this analysis we considered iOS 6, 8, and 9 devices, as well as Mac clients running OS X 10.10.3, 10.10.5, and 10.11.1.

service, which is operated by Apple using both their own servers and virtual servers provisioned on Amazon AWS, Microsoft Azure, and Google’s Cloud Platform.

Identity and registration The basic unit of identity in iMessage is the iCloud account name, which typically consists of an email address or phone number controlled by the user. End-user devices are registered to the iCloud service by associating them with an account. The mapping between client devices and accounts is not one-to-one: a single account may be used across multiple devices, and similarly, multiple accounts can be associated with a single device. We give further information about the registration process in Appendix A.

Message encryption and decryption To transmit a message to some list of Recipient IDs, the Sender’s iMessage client first contacts the IDS to obtain the public key(s) PK_1, \dots, PK_D and a list of APNs push tokens associated with the Sender and Recipient identities.³ It then encodes the Sender and Recipient addresses and plaintext message into a binary plist key-value data structure and compresses this structure using the gzip compression format. The client next generates a 128-bit AES session key K and encrypts the resulting compressed message using AES-CTR with $IV = 1$. This produces a ciphertext c , which is next partitioned as $c = (c_1 \| c_2)$ where c_1 represents the first 101 bytes of c . The Sender parses each PK_i to obtain the public encryption key $pk_{E,i}$ and for $i = 1$ to D , calculates $C_i = \text{RSA-OAEP}(pk_{E,i}, K \| c_1)$ and a signature $\sigma_i = \text{ECDSASign}(sk_S, C_i \| c_2)$. For each distinct push token received from IDS, the Sender transmits (C_i, c_2, σ_i) to the APNs server. This process is illustrated in Figure 1.

For each ciphertext, the APNs service delivers the tuple $(ID_{\text{sender}}, ID_{\text{recipient}}, C_i, c_2, \sigma_i)$ to the intended destination. The receiving device contacts IDS to obtain the Sender’s public key PK , parses for the signature verification key vk_S , then verifies the signature σ . If verification succeeds, it decrypts C_i to obtain $K \| c_1$, reconstructs $c = (c_1 \| c_2)$ and decrypts the resulting AES-CTR ciphertext using K . It decompresses the resulting gzip ciphertext, parses the resulting plist to obtain the list of Recipient IDs, and verifies that each of ID_{sender} and $ID_{\text{recipient}}$ are present in this list. If any of the preceding checks fail, or if the Recipient is unable to parse or decompress the resulting message, the receiving device silently aborts processing.

³This list includes one entry for each device registered to each Sender and Recipient ID. The Messages client encrypts the message with each Sender public key to ensure that message transcripts can be read across all of the Sender’s devices.

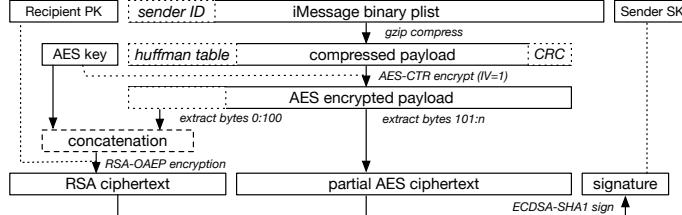


Figure 1: The iMessage encryption mechanism. From the top, each iMessage is encoded in a binary plist key/value structure. The structure encodes a list of Sender and Recipient account identifiers, as well as the message contents. This payload is subsequently gzip compressed, and encrypted under a freshly-generated 128-bit message key using AES in CTR-mode. The AES key and the first 101 bytes of the AES ciphertext are concatenated and are encrypted to each Recipient’s public key using RSA-OAEP. The remaining bytes of the AES ciphertext are concatenated to the RSA ciphertext and the result is signed using ECDSA under the Sender’s registered signing key.

Attachments and long messages For long messages and messages containing file attachments (e.g., images or video), iMessage delivers the encrypted data using a separate mechanism. First, the client generates a 256-bit AES key K' and encrypts the attached data using AES in CTR mode. It next uploads the resulting encrypted document to Apple’s iCloud service and obtains a unique `icloud.com` URL and an access token for the attachment. In the course of this process, the iCloud service may redirect the client to upload the encrypted file to a third-party storage server operated by an outside provider such as Amazon, Microsoft or Google. Having uploaded the attachment, the client now constructs a standard iMessage plist containing the URL and access token, the key K' , and a SHA1 hash of the encrypted document. This plist, which may also include normal message text, is encrypted and transmitted to the Recipient using the standard message encryption mechanism. Upon receiving and decrypting the message, the Recipient downloads the attachment using the provided URL and access token, verifies that the provided hash matches the received attachment, and decrypts the attachment using K' .

3 Security goals & Threat model

Apple has stated that iMessage is an end-to-end encryption protocol that should be secure against all attackers that do not have control of Apple’s network. We base our threat model on a recent survey on secure messaging by Unger *et al.* [38]. This threat model includes the following attackers:

Local Adversary. This includes an attacker with control over local networks, either on the Sender or Recipient side of the connection.

Global Adversary. An attacker controlling large segments of the Internet, such as powerful nation states

or large Internet service providers.

Network operator. Apple operates centralized infrastructure for both public key distribution and message transmission/storage. Potential adversaries include Apple, a government, or a malicious party with access to Apple’s servers.

Each of these attackers may be active or passive. A passive attacker simply observes traffic and does not seek to alter or inject its own messages. An active attacker may issue arbitrary messages to any party. In many cases, these adversary classes may interact. As in [38] we assume that adversaries also have access to the messaging system and can use the system to register accounts and transmit messages as normal participants. We also assume that the endpoints in the conversation are secure.

4 High-level Protocol Analysis

An initial analysis of the iMessage specification shows that the protocol suffers from a number of defects. In this section we briefly detail several of these limitations. In the following sections we focus on specific, exploitable flaws in the encryption mechanism.

Key server and registration iMessage key management uses a centralized directory server (IDS) which is operated by Apple. This server represents a single point of compromise for the iMessage system. Apple, and any attacker capable of compromising the server, can use this server to perform a man-in-the-middle attack and obtain complete decryption of iMessages. The current generation of iMessage clients do not provide any means for users to compare or verify the authenticity of keys received from the server.

Of more concern, Apple’s “new device registration” mechanism does not include a robust mechanism for notifying users when new devices are registered on their



Figure 2: Example of a simple ciphertext replay.

account. This mechanism is triggered by an Apple push message, which in turn triggers a query to an Apple-operated server. Our analysis shows that these protections are fragile; in Appendix A we implement attacks against both the key server and the new device registration process.

Lack of forward secrecy iMessage does not provide any forward secrecy mechanism for transmitted messages. This is due to the fact that iMessage encryption keys are long-lived and are not replaced automatically through any form of automated process. This exposes users to the risk that a stolen device may be used to decrypt captured past traffic.

Moreover, the use of long term keys for encryption can increase the impact of other vulnerabilities in the system. For example, in §5, we demonstrate an active attack on iMessage encryption that exposes current iMessage users to decryption of past traffic. The risk of such attacks would be greatly mitigated if iMessage clients periodically generated fresh encryption keys. See §7 for proposed mitigations.

Replay and reflection attacks The iMessage encryption protocol does not incorporate any mechanism to prevent replay or reflection of captured ciphertexts, leading to the possibility that an attacker can falsify conversation transcripts as illustrated in Figure 2. A more serious concern is the possibility that an attacker, upon physically capturing a device, may replay previously captured traffic to the device and thus obtain the plaintext.

Lack of certificate pinning on older iOS versions iMessage clients interact with many Apple servers. As of December 2015, Apple has activated certificate pinning on both APNs and ESS/IDS connections in iOS 9 and OS X 10.11. This eliminates a serious attack noted by Raynal *et al.* [34] in which an MITM attacker who controls the Sender’s local network connection and possesses an Apple certificate can intercept calls to the ESS/IDS key server and substitute chosen encryption keys for any Recipient (see Appendix A for further details). We note that

devices running iOS 8 (and earlier) or versions of OS X released prior to December 2015 may still be vulnerable to such attacks. For example, at the time of our initial disclosure in November 2015 to Apple, pinning was not present in OS X 10.11.

Non-standard encryption iMessage encryption does not conform to best cryptographic practices and generally seems *ad hoc*. The protocol (see Figure 1) insecurely composes a collection of secure primitives, including RSA, AES and ECDSA. Most critically, iMessage does not use a proper authenticated symmetric encryption algorithm and instead relies on a digital signature to prevent tampering. Unfortunately it is well known that in the multi-user setting this approach may not be sound [21]. In the following sections, we show that an on-path attacker can replace the signature on a given message with that of another party. This vulnerability gives rise to a *practical* chosen ciphertext attack that recovers the full contents of some messages.

5 Attacks on the Encryption Mechanism

In this section we describe a practical attack on the iMessage encryption mechanism (Figure 1) that allows an attacker to completely decrypt certain messages.

5.1 Attack setting

Our attack assumes that an adversary can recover encrypted iMessage payloads and subsequently access the iMessage infrastructure in the manner of a normal user. The first requirement implies one of two conditions: in condition (1) the attacker is on-path and capable of intercepting encrypted iMessage payloads sent from a client to Apple’s Push Notification Service (APNs) servers. Since the APNs protocol employs TLS to secure connections between the client and APNs server, this attacker must possess some means to bypass the TLS encryption layer; we discuss TLS interception in more detail in Appendix B. In condition (2) the attacker can recover iMessage ciphertexts from within Apple’s network. This requires either a compromise of Apple’s infrastructure, a rogue employee, or legal compulsion. Figure 3 describes the network flow of a single iMessage, along with potential attacker locations.

5.2 Attack overview

There are two stages of the attack. The first exploits a weakness in the design of the iMessage encryption composition: namely, that iMessage does not properly authenticate the symmetrically encrypted portion of the message payload. In a properly-designed composition,

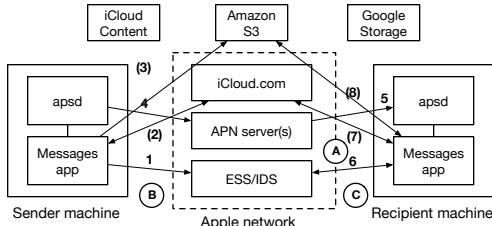


Figure 3: The process of sending an iMessage through the APNS network. The steps are as follows: (1) The Sender contacts ESS/IDS to obtain the public keys for each Recipient; (2) (*optional*) the Sender contacts iCloud to upload an attachment; (3) (*optional*) the Sender uploads the encrypted attachment to an outside storage provider as directed by iCloud; (4) the Sender’s apsd instance transmits the encrypted iMessage payload to Apple’s APNs server; (5) Apple delivers the payload to a Recipient; (6) the Recipient contacts ESS/IDS to obtain the Sender’s public key; (7) (*optional*) the Recipient contacts iCloud if an attachment is present; (8) (*optional*) the Recipient downloads the encrypted attachment from an outside storage provider. Potential attacker locations are labeled A, B and C.

this section of the ciphertext would be authenticated using a MAC in generic composition [14] or via an AEAD mode of operation. Apple, instead, relies on an ECDSA signature to guarantee the authenticity of this ciphertext. In practice, a signature is insufficient to prevent an attacker from mauling the ciphertext since an on-path attacker can simply replace the existing signature with a new signature using a signing key from an account controlled by the attacker. In practice, the actual attack is slightly more complex; the first phase includes additional operations to defeat a countermeasure in the decryption mechanism, which we discuss below.

The second stage of the attack leverages the ability to modify the AES ciphertext (specifically, the section not contained within the RSA ciphertext). This phase consists of an adaptive chosen ciphertext attack exploiting the structure of the underlying plaintexts. The attack repeatedly modifies the ciphertext and sends it to either the Sender or a Recipient for decryption. If the attacker can determine if decryption and parsing were successful on the target device, she can gradually recover the underlying iMessage payload.

The attack specifics are reminiscent of Vaudenay’s padding oracle attack [40], but relies on the usage of *compression* within the iMessage protocol. Specifically, our attack takes advantage of the 32-bit CRC checksum, computed over the pre-compressed message, incorporated into gzip compressed ciphertexts. Since CRCs are

linear under XOR but the compression function is not, we can verify guesses about message content by editing the compressed, encrypted message and testing if the corresponding correction to the CRC results in a valid message.⁴

5.3 A format oracle attack for gzip compression

The gzip format [23] uses DEFLATE compression which itself combines LZ77 [41] and Huffman coding to efficiently compress common data types. The format supports both static and dynamically-generated Huffman tables, though most encoders use dynamic tables for all but the shortest messages. To compress a message, a CRC32 C is calculated over the uncompressed input. Next, the encoder identifies repeated strings and replaces each repeated instance with a tuple of the form $\langle length, backwards\ distance \rangle$, where distance indicates the relative position of the previous instance of the string. The input is encoded using an alphabet of 286 symbols, comprising the 256 byte literals, an end-of-block (EOB) symbol, and 29 string replacement length values.⁵ If dynamic generation is selected, a Huffman table T is calculated using the resulting text as a basis (for static tables, $T = \epsilon$), and the text is Huffman coded into a string of variable-length symbols $S = (s_1, \dots, s_N)$ where string replacement symbols are internally partitioned into a pair $\langle length, distance \rangle$. The resulting compressed message consists of (T, S, C) . On decompression the process is reversed and the CRC of the resulting string is compared to C . If any step fails, the decompressor outputs \perp .

Attack intuition. Our attack assumes that the attacker has intercepted a gzip compressed message encrypted using an unauthenticated stream cipher and that we have access to a decryption oracle that returns 1 if and only if the message decrypts and successfully decompresses. Our goal is to recover a substantial fraction of the plaintext message.

For clarity, we assume the attacker knows the Huffman table T and the length in bits L of the uncompressed input. We further assume the attacker knows the exact location in the ciphertext corresponding to some (unknown) ℓ -bit Huffman symbol s that she wishes to recover, as well as the position of the corresponding decoded literal in the uncompressed text. These are simplifying assumptions and we will remove them as we proceed.

Given a ciphertext c , our attack works by first selecting a mask $M \in \{0, 1\}^\ell, M \neq 0^\ell$ and perturbing the ci-

⁴Were the compression function and CRC both linear, the edit to the CRC and compressed text would always cancel.

⁵A separate Huffman table is used to encode backwards distances.

phertext such that the underlying symbol s will decrypt to $s' = s \oplus M$. This is done by *xoring* M into the ciphertext at the appropriate location. Let $\text{decode}(T, s)$ and $\text{decode}(T, s')$ represent the Huffman decoding of s and s' respectively, and let repeats be a boolean variable that is true if and only if s (resp. s') is repeated subsequently via a DEFLATE string replacement reference. The potential values of these three variables can be categorized into the following seven cases:

Case	$\text{decode}(T, s)$	$\text{decode}(T, s \oplus M)$	repeats
1	[0, 255]	[0, 255]	False
2	[0, 255]	[0, 255]	True
3	[0, 255]	[256, 285]	(either)
4	[0, 255]	\perp	(either)
5	[256, 285]	[0, 255]	(either)
6	[256, 285]	[256, 285]	(either)
7	[256, 285]	\perp	(either)

In the following paragraphs, we consider the outcome of our experiment for each of the cases above.

CASE 1: In this case, when the attacker submits the mangled ciphertext to the decryption oracle, the oracle will internally decode a result that differs from the original input string in exactly one byte position: the position corresponding to symbol s' . However, with overwhelming probability, the CRC C' of the decompressed string will not match C and cause the oracle to output 0.

Because CRC is linear under XOR, the attacker may correct the encrypted value C by further mauling the ciphertext. Let d indicate the bit position of the symbol associated with s (resp. s') in the decoded message. For each $i \in \{0, 1\}^8$ the attacker *xors* the string $\bar{C} = \text{CRC}(0^d || i || 0^{L-d}) \oplus \text{CRC}(0^L)$ with the ciphertext at the known location of C and submits each of the resulting ciphertexts for decryption. Since we have that $\text{decode}(T, s') \in [0, 255]$, one of these tests will always result in a successful CRC comparison.

Upon receiving a successful result from the decryption oracle, the attacker now examines the Huffman table T to identify candidate symbols s for which relation $\text{decode}(T, s \oplus M) = \text{decode}(T, s) \oplus i$ holds. If the attacker cannot identify a unique solution for s , she may select a new $M' \neq M \neq 0^\ell$ and repeat the procedure described above until she has uniquely identified s . The attacker can now increment her position in the ciphertext by ℓ bits and repeat this process to obtain the next plaintext symbol.

If this experiment is unsuccessful, it indicates that the ciphertext is not in Case 1 from the above table. To determine which case applies, the attacker must conduct additional experiments as described below. Sometimes recovery of the symbol s will not be feasible at all; when this occurs, the attacker must simply continue to the next symbol in S . Occasionally, the adversary may still be able to recover s at some additional cost.

CASE 2: In this case, the symbol represented by s (resp. s') is referenced by one or more subsequent instances of DEFLATE string repetition. The practical impact is that modifying s will produce an identical alteration at two or more positions in the decoded string and with high probability none of the experiments indicated for Case 1 will succeed.

In some circumstances, it may be cost effective for the attacker to skip s and simply move on to the next symbol in S . Alternatively, the attacker can experimentally modify the CRC to indicate the same alteration at *all* positions that could be affected by modifying s . Since the attacker does not know the locations at which s is repeated or the number of such locations, this requires the attacker to submit many candidate ciphertexts to the oracle, one for each possible set of locations where s may repeat. In the event that s (resp. s') is repeated only once, this requires the attacker to issue $2^8 \cdot (L - d)/8$ queries to the oracle (one for each value of i and for each possible location for the repeated value of s'). This may be feasible for reasonably short strings.

CASES 3-4: In these cases, the original decoding of s was a byte literal, but the decoding of s' is either an invalid symbol or a special symbol (EOB or string replacement symbol). The former case always results in decompressor failure, while the latter will typically cause the decoded string to differ from the original input at multiple locations, resulting (with high probability) in a CRC comparison failure that will not be corrected by the procedure described above.

To address these cases, the attacker may select a new mask $M' \neq M \neq 0^\ell$ and repeat the complete experiment described above. Depending on the structure of the Huffman table T , and provided that $s \in [0, 255]$, the new result $s \oplus M'$ may produce an outcome that satisfies the conditions of cases (1) or (2).⁶

CASES 5-7: These cases occur when the original symbol represented by $\text{decode}(T, s)$ is a string replacement or EOB symbol. In most instances, replacing s with $(s \oplus M)$ produces a decoded string that differs from the original in many positions, making it challenging for the attacker to repair the CRC. If s decodes to a string replacement token, and the replacement reference points to a location that the attacker has already recovered, it may be possible for the attacker to detect the alteration using the technique described under Case 2. Otherwise the attacker must skip s and move on to the next symbol in S .

Recovering the unknowns. The procedure described so far requires the attacker to know the Huffman table T , the

⁶In principle, this approach might require as many as $2^8 \cdot 2^{|M|} = 2^{8+\ell}$ decryption queries to obtain a successful result, or rule out these cases. In practice, however, the number of candidate mask values M' is likely to be much more limited.

length of the uncompressed message L , the location and length of the symbol s , and the byte index of the corresponding decompressed literal. In practice many of these quantities may be determined experimentally by iterating through candidate values for L, ℓ, k and the symbol position. This requires the attacker to issue many candidate decryption requests until one succeeds. In the case of iMessage attachment messages, the length L is fixed and an attacker can generate a representative corpus of messages offline and easily estimate the other parameters *without* oracle queries.

Recovering the Huffman table is more challenging. If the message is encoded using a static table, then the table is known to the attacker. However, if T is dynamically generated, then the attacker learns only the relation $\text{decode}(T, s \oplus M) = \text{decode}(T, s) \oplus i$, but has no clear way of learning s or $\text{decode}(T, s)$. Nonetheless, it might still be possible to recover enough information from these relations to recover the value of the underlying literals.

However, in iMessage this proves unnecessary as we take advantage of iMessage’s structure to recover a large fraction of the dynamic table T . iMessage payloads containing attachments embed a URL within the encrypted message, requests to which can be monitored (described below). In this way, we learn the file path and/or hostname indicated by the plaintext URL within each ciphertext. Given this information, and by mauling individual symbols s contained within the URL string, the attacker can recover the value $\text{decode}(T, s \oplus M)$ for many different values of M . This allows the attacker to identify a relative-distance map of a portion of the Huffman tree. This proves sufficient to recover much of the Huffman table T .

Detecting successful decryption. Our attack assumes that the attacker can detect successful decryption of a modified ciphertext. To simplify this assumption, we focused on messages containing attachments, such as images and videos. These messages include a URL for downloading the attachment payload, as well as a 256-bit AES key to be used in decrypting the attachment. When an iMessage client correctly decrypts such a message, it automatically initiates an HTTPS POST request to the provided URL. A local network attacker can view (and intercept) this request to determine whether decryption has occurred. Moreover, if the attacker blocks the connection, the device will retry several times and then silently abort. Since the client provides no indication to the user that a message has been received, this admits silent decryption of ciphertexts.

This technique can also be extended to situations where the attacker is not on the target device’s local network. By mauling the URL field to change the requested hostname (e.g., from `icloud.com` to a domain that the

attacker controls), the attacker can simply direct the target device to issues HTTPS to a machine that the attacker controls. This allows the attacker to conduct the attack remotely by transmitting ciphertexts through Apple’s APNs network, at which point she obtains the full HTTPS POST request from the target device. Since the attacker controls the request domain, there is no need to MITM the TLS connection.⁷

5.4 An Attack on Attachment Messages

Having provided an overview of the attack components, we will describe each individual step of the complete attack. This attack scenario assumes that a target Sender has transmitted an attachment-bearing message to one or more online receivers, and the attacker has the ability to monitor the local network connection (and intercept TLS connections) on one of the Sender or Recipient devices.

Step 1. Removing and replacing the iMessage signature.

Each iMessage is authenticated using an ECDSA signature, formulated using the private key of the iMessage Sender. This signature prevents the attacker from directly tampering with the message. However, a limitation of using signatures for authenticity is that they do not prevent ciphertext mauling when an attacker controls another account in the system. An attacker who intercepts a signed iMessage may simply remove the existing signature from the message and re-sign the message using a different key, corresponding to a separate account that the attacker controls.⁸ The attacker now transmits the resulting encrypted payload, signed and delivered as though from a different Sender address. The signature replacement process is illustrated in Figure 4.

In practice, simply replacing the signature on a message proves insufficient. In iMessage, a full list of Sender and Recipient addresses is specified both in the unencrypted metadata for the message *and* in the encrypted message payload. Upon decrypting each message, iMessage clients verify that the message was received from one of the accounts listed in the Sender/Recipient list, and silently abort processing if this condition does not hold.⁹ While it is trivial to replace the unencrypted Sender field, replacing encrypted envelope information is more challenging. Fortunately, in most cases this field of the iMessage plist is contained within the malleable

⁷The current versions of Apple’s Messages client do not enforce that this URL contains `icloud.com` and will connect to any hostname provided in the URL. Similarly, the Messages client does not pin certificates for the HTTPS connection.

⁸On Mac OS X, iMessage signing keys are readily accessible from the Apple Keychain.

⁹Based on our experiments, the participant list does not appear to be ordered, or to distinguish between Sender and Recipients. It is sufficient that the Sender identity appears somewhere in this list.

AES-CTR ciphertext, and we are able to alter the contents of the Sender/Recipient list so that it contains the identity of the replacement Sender account.

Step 2. Altering the Sender identity.

To alter the Sender identity, the attacker must selectively maul the AES-CTR ciphertext to change specific bytes of the Sender/Recipient plist field to incorporate the new Sender identity she is using to transmit the mauled ciphertext. This is challenging for several reasons.

First, the initial 101 bytes of the AES ciphertext are stored within the RSA-OAEP ciphertext, which is strongly non-malleable. Thus we are restricted to altering the subsequent bytes of the ciphertext. Fortunately, the binary plist key-value data structure is *top heavy*, in that it stores a list of all key values in the data structure prior to listing the values associated with each key. In practice, this ensures that the relevant Sender identity appears some distance into the data structure. Moreover, the application of gzip compression produces additional header information, including (in many cases) a dynamic Huffman table. In all of the cases we observed, the symbols encoding the Sender identity are located subsequent to the first 101 bytes, and are therefore not included within the OAEP ciphertext.

The use of gzip compression somewhat complicates the attack. Rather than mauling uncompressed ASCII bytes, the attacker must alter a set of compressed Huffman symbols which have been encoded using a (dynamically-generated) table T that the attacker does not know. Fortunately, the attacker knows the original identity of the Sender, as this value is transmitted in the unencrypted apsd metadata. Moreover, in all iMessage clients that we examined, the Sender identity is transmitted as the first string in the Sender/Recipient list, which – due to iMessage’s predictable format – appears in a relatively restricted range of positions within the ciphertext. Even with this knowledge, altering the Sender ID involves a large component of guessing. The attacker first estimates the location of the start of the Sender/Recipient list, then selectively mauls the appropriate portions of the AES ciphertext, while simultaneously updating the CRC to contain a guess for the modified (decoded) symbol. This is a time consuming process, since the attacker must simultaneously identify (1) the appropriate location in the ciphertext for the symbol she wishes to modify and (2) a modification that causes the symbol to change to the required symbol. The target device will silently ignore any incorrect guesses and will proceed with attachment download only when the mauled Sender ID in the plist is equal to the Sender ID from which the attacker is transmitting.

To simplify the attack, the attacker may restrict her at-

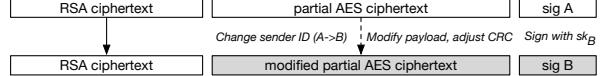


Figure 4: Modifying the partial AES ciphertext, including the Sender ID and CRC, and replacing the signature with a new signature corresponding to an account (and signing key) we control.

tention to addresses that differ from the original Sender ID in at most one symbol position. This is accomplished by registering new iCloud addresses that are “one off” from the target Sender identity. To increase the likelihood that we will succeed in altering the Sender account to match one that we have selected, we register multiple new Sender identities that are near matches to the original identity. For each attempt at mauling the ciphertext, we must also “repair” the CRC by guessing the effect of our changes on the decompressed message.

In our experiments, we found that an email address of the form `abcdef@icloud.com` could be efficiently modified to a new account of the form `abcdef@i8loud.com` in approximately 2^{10} decryption queries to a target device.¹⁰ Since Huffman tables vary between messages, we cannot mutate every message to the same domain, and thus we need to control several variants of `icloud.com` for this strategy to be successful in all cases. Fortunately, the edits are predictable and our simulations indicate that we require only one domain to recover most messages.

A side effect of this modification is that, due to string replacement in gzip, the attachment URL is simultaneously altered to point to `i8loud.com`, which means that attachment HTTPS POST requests are sent to a computer under our control. This makes it possible to conduct the attack remotely.

Step 3. Recovering the Huffman table. Given the ability to intercept the attachment request POST URL to `icloud.com`, we now recover information about the dynamic Huffman tree T used in the message. The attachment path consists of a string of alphanumeric digits, which in most instances are encoded as Huffman symbols of length $\ell \in [4, 8]$.

By intercepting the HTTPS connection to `icloud.com`, the attacker can view the decoded URL path and systematically maul each Huffman symbol in turn, repairing the CRC using the technique described in the previous subsection. This allows the attacker to gradually recover a portion of the Huffman tree (Figure 5). In practice, the attacker is able to recover only a subset of the tree, however, because the iMessage client will silently fail on any URL that

¹⁰These email addresses are examples and not the real email addresses we used in our experiments.

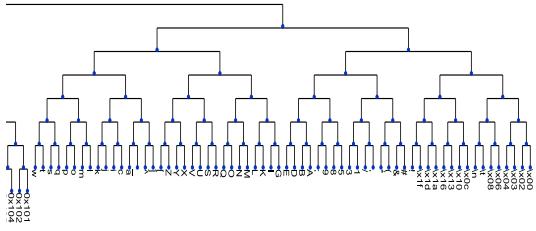


Figure 5: Fragment of a Huffman tree from an attachment iMessage.

contains characters outside the allowed URL character set.¹¹ Fortunately this set includes most printable alphanumeric characters.

Our implementation recovers a portion of the Huffman tree that is sufficient to identify the characters in the set $0 - 9, A - F$. Our experiments indicate that this phase of the process requires an average 2^{17} decryption requests and a maximum of 2^{19} .

Step 4. Recovering the attachment encryption key. When an iMessage contains an attachment, the message embeds a 256-bit AES key that can be used to decrypt the attachment contents. This key is encoded as 64 ASCII hexadecimal characters and is contained within a field named `decryption-key`. An attacker with oracle access to a target device, and information on the Huffman table T , can now systematically recover bytes from this key. Upon recovering the key, they can use the intercepted HTTPS request information to download the encrypted attachment and decrypt it using the recovered key.

The approach used in recovering the attachment key is an extension of the general format oracle attack described above. The attacker first searches the ciphertext to identify the first position of the decryption key field. The attacker identifies a mask M (typically a single or double-bit change to the ciphertext) that produces a change in the decoded message at the first position of the encryption key, which is known due to the predictable structure of attachment messages. To identify this change, the attacker “fixes” the CRC to test for each possible result from the decryption key, then learns whether the decryption/decompression process succeeds. To obtain the full key, the attacker repeats this process for each of the 64 hexadecimal symbols of the encryption key.

This process does not reliably produce every bit of the key, due to some complications described in the general attack description above. Principal among these is the fact that some Huffman symbols represent string replacement tokens rather than byte literals. While it seems

counterintuitive to expect repeated strings within a random key, this occurrence is surprisingly common due to the fact gzip will substitute even short (3 digit) strings. Indeed, on average we encounter 1.9 three-digit repetitions within each key. In this case, we attempt to identify subsequent appearances of the symbol by guessing later replacement locations. If this approach fails, our approach is to simply ignore the symbol and experimentally move forward until we reach the next symbol.

While it is possible to recover a larger fraction of the symbols in the message by issuing more decryption queries (see §6 for a discussion of the tradeoffs), in many cases it is sufficient to simply guess the missing bits of the key offline after recovering an encrypted attachment. In practice, the entropy of the missing sections is usually much lower than would be indicated by the number of missing bits, since in most cases the replacement string is drawn from either the URL field or earlier sections of the key, both of which are known to the attacker.

Step 5. Recovering the message contents. Each attachment message may also contain message text. This text can be read in a manner similar to the way the key is recovered in the previous step, by mauling the message portion of the text and editing the CRC appropriately. This approach takes slightly more effort than the hexadecimal key recovery step, due to the higher number of potential values for each Huffman symbol in the message text.

6 Implementation and Evaluation

6.1 Estimating attack duration

To validate the feasibility of the attack described in §5.4, we implemented a prototype of the gzip format oracle attack in Python and executed it against the Messages client on OS X 10.10.3. Our attack successfully recovered 232 out of 256 key bits after 2^{18} decryption queries to the target device. The main challenge in running the attack was to determine the correct timeout period after which we can be confident that a message has not been successfully decrypted. This timeout period has a substantial impact on the duration of the attack, as we describe below.

Experimental Setup To deliver iMessage payloads to the device, we customized an open-source Python project called `pushproxy` (hereinafter called the proxy) and used it to intercept connections from the device to Apple’s APNs server [3]. This approach models an attacker who can either impersonate or control Apple’s APNs servers. While our attack assumed local network interception and did not send messages through Apple’s

¹¹iMessage does not perform URL coding on disallowed characters.

servers, we note that if an attacker is able to capture messages in transit (by bypassing TLS or by compromising Apple’s servers), the remainder of the attack can in principle be conducted remotely (see the end of §5.3 for details). For ethical and legal reasons, we explicitly chose not to test attacks that relayed messages via Apple’s production servers. Thus all of our attacks were conducted via a local network.

To address the use of TLS on apsd connections, we configured our modified proxy with a forged Apple certificate based on a CA root certificate we created and change `/etc/hosts` to redirect APNs connections intended for Apple towards our local proxy. We generate the forged certificate by installing our root CA on the target system.¹²

To monitor and intercept attachment download requests, we configured an instance of a TLS MITM proxy (`mitmproxy`) using our self-signed root certificate to intercept all outbound requests from the device made via HTTP/HTTPS. When the target device receives an attachment message, it makes two HTTPS POST requests to `{0, ..., 255}-content.icloud.com`. Based on the result of these requests, the device issues a second HTTP GET request to download the actual attachment. In our experiments we block both of the POST requests, ensuring that no indication of the message processing is displayed by the Messages client. For each oracle query, the attack code waits for `mitmproxy` to report an attachment POST request as defined above or, after a set time out, assumes the oracle query resulted in a failed message.

Finally, we created an iMessage account for the attacker that is a single-character edit of the sender’s address (e.g. if the sender is `alice@example.com`, the attacker might be `clice@example.com`). We only generate one such account for the edit we expect to be successful, although a real attacker might register a large corpus of iMessage accounts and thus increase the success probability of this phase of the attack.

Verifying the existence of the oracle To ensure that iMessage behavior is as expected, we conducted a series of tests using hand-generated messages to determine if we were able to detect decryption success or failure on these messages. Our results were sufficient to confirm the vulnerability of §5 and verify iMessage’s behavior sufficiently well that we could construct a simulated oracle for our experiments of §6.2.

Estimating the timeout for failed queries The main goal of our experiment was to determine the maximum

¹²Since OS X 10.10.3 does not include certificate pinning for APNs connections, this allowed us to intercept and inject iMessage ciphertexts.

timeout period after which we can determine that the device has been unable to successfully decrypt and process a message. To determine this, our attack queries the gzip format oracle by sending a candidate message and waiting until it either sees a resulting attachment download (in which case the message decrypted) or some timeout passes. Too long of a timeout results in unreasonable runtimes and too short of a timeout produces false negatives, which lead to incorrect key recovery.

Small scale experiments proved unable to reliably estimate the maximum timeout: the observed wait time distribution seemingly has a long tail and may be dependent on load not encountered in small experiments (e.g. due to failed decryptions). Using the full attack code to find the max timeout, on the other hand, is impractical, since we must run 2^{18} queries, each lasting as long as the timeout. This would take between 18 hours and 3 days depending on the timeout duration we wish to test.

In order to estimate the correct timeout, we ran our attack on the device in tandem with a local instance of the format oracle which, using the recipient’s private key, also decrypts the message and emulates iMessage’s behavior. If the candidate message fails to decrypt against the local oracle, we use a short (400ms) timeout period. If the candidate message decrypts successfully on this local oracle, then we wait an unbounded amount of time for the oracle query and record the necessary delay. We stress that this local-oracle approach was used only to speed up the process of finding the maximum delay; the full attack can be conducted *without* knowledge of the private key.

Results We ran our main experiment on a real message intercepted using the proxy. It recovers 232 out of 256 key bits in 2^{18} queries and took 35 hours to run. The maximum observed delay between a query and the resulting download request was 903ms, while the average was 390ms with a standard deviation of 100ms. Based on this data, and without considering further optimizations, we estimate that the full attack would require approximately 73 hours to run if we naively used 1 second as the timeout.

Optimizing runtime The obvious approach to optimizing our attack is to reduce the timeout period to the minimum period that iMessage requires to successfully process and queue a message. Through experiments, we determined this to be approximately 400ms. Thus one avenue to optimizing the experiment is to reduce the timeout period for all messages to 400ms, using the assumption that a successful experiment may result in a “late” download. Since we would not be able to neatly determine the specific message query that occasioned the

download, we would need to temporarily increase the delay period and “backtrack” by repeating the most recent, e.g., 10 queries to determine which one caused the download. Because the issue is patched and further exploration difficult, we have elected not to implement these optimizations.

Because successful queries are quite sparse,¹³ this does not meaningfully affect the number of queries needed for the attack. In our estimation, these techniques will reduce the cost of the full attack down to 35 hours and requires only straightforward modifications to our proof of concept code.

A second optimization is to run the attack against multiple devices with attack queries split and conducted in parallel against them. For n devices, the attack time is reduced by approximately a factor of n . As many users may have 2 or 3 devices, this can offer substantial reductions.

Finally, we can reduce the raw number of queries needed to mount the attack by refining the gzip-oracle attack techniques. In particular, we can reduce the number of queries needed to recover the Huffman table by inferring the structure of the tree from the partial information we have and from the observation that the Huffman trees fall within a fairly limited range of distributions. In particular we note that for the Huffman trees used in gzip, recovering the symbol lengths alone is sufficient to recover the tree. An approach drawing from techniques in machine learning to recover the Huffman table given only a few queries, the distribution of such tables, and known partial information could offer substantial improvements. We leave a full exploration of these optimizations to future work.

6.2 Simulation results

Although we have conducted our attack on iMessage, we have not explored its effectiveness with a large range of messages. Given the time it takes to run an experiment, doing so is prohibitive. We opt instead to simulate our results.

Simulation To evaluate the overall effectiveness of our format oracle attack, we constructed a simulated message generator and decryption oracle. Messages produced by our generator are distributed identically to real attachment-bearing messages, but contain randomly-generated strings in place of the filename, URL path, Sender and Recipient addresses, decryption key, and “signature” (hash) fields. The decryption oracle emulates the iMessage client’s parsing of the inner binary plist.

¹³Out of the 2^{18} , only 418 were successful.

For performance, it skips encryption and decryption.¹⁴ Decompression is done using Python’s `gzip` module, which is a wrapper around `zlib`. We experimentally validate the oracle’s correctness against the transcript of a real attack and against separate messages.

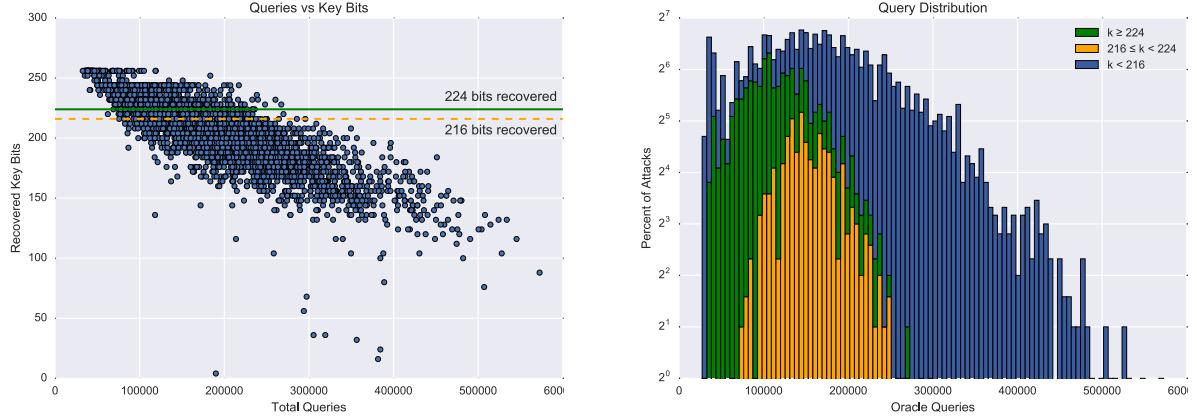
Results We ran our simulated attack on a corpus of 10,000 generated messages and show the results in Figure 6. In all cases, our experiments completed in at most 2^{19} queries, with an average of approximately 2^{17} queries. For 34% of the experiments we ran, our attack was able to recover ≥ 216 bits of the attachment AES key. For 23% of the messages we experimented with, we recovered ≥ 224 bits of the key, enabling rapid brute-force of the remaining bits on commodity hardware.¹⁵

Optimizing success rate Many of the failures we experience in key recovery are caused by issues with string repetition. Recall that repeated substrings in a message are compressed in gzip by replacing all subsequent repetitions of the substrings with a backwards-pointing reference. As a result, editing the canonical location of a substring in the compressed message may cause similar changes to future instances of the same substring in the decompressed message. Our CRC correction for a given location fails to compensate for these later changes because we simply do not know where in the uncompressed message the second instance of the substring appears. As a result, our current attack simply skips these bits.

However, we can address this weakness with only a modest increase in the number of oracle queries. By scanning through the remaining bytes and applying the same CRC correction at each subsequent location in the uncompressed message, we can identify the location of the subsequent instances of the substring. This is efficient mainly for strings that are repeated twice, but our experiments indicate this is the most common case. Note that we do not need to scan through the entire message. As a result of the particular format of the messages, there are only a few points where we can get duplicates: most of the message is in lowercase letters or non-printable characters, whereas the `decryption-key` and `mmcs-url` field (i.e. the locations where repeats cause the most serious issues) are upper case alpha-numeric and hence will not contain repeats from the majority of the other fields. For the experiments described above,

¹⁴Our implementation prevents the attacker from modifying the first 101 bytes of the message, as those are normally contained within the RSA ciphertext. Additionally, the oracle enforces that the alleged Sender identity is included within the `plist`, which is a condition enforced by iMessage.

¹⁵Experiments on an inexpensive Intel Core i7 show that we can recover 32 missing key bits in approximately 7 minutes using an AES-NI implementation. Therefore recovering 40 missing key bits should take approximately 28 hours on a single commodity desktop.



(a) Number of queries vs number of recovered key bits. The orange dashed line represents 216 bits recovered, the solid green line 224.

(b) Distribution of attack length, measured in queries. The high concentration of attacks near zero is due to a rapid failure when it fails to edit the sender email.

Figure 6: Simulation results for the attachment recovery attack.

this would result in a 14% increase in the number of messages for which we can recover 224 bits.

7 Mitigations

Our main recommendation is that Apple should replace the entirety of iMessage with a messaging system that has been properly designed and formally verified. However, we recognize this may not be immediately feasible given the large number of deployed iMessage clients. Thus we divide our recommendations into short-term “patches” that preserve compatibility with existing iMessage clients and long-term recommendations that require breaking changes to the iMessage protocol.

7.1 Immediate mitigations

Duplicate RSA ciphertext detection. The attacks we described in §5 are possible because the unauthenticated AES encryption used by iMessage is malleable and does not provide security under adaptive chosen ciphertext attack, unlike RSA-OAEP encryption [15]. Maintaining a list of all previously-received RSA ciphertexts should prevent these replay and CCA attacks without the need for breaking changes in the protocol. Upon receiving a stale RSA ciphertext, the Recipient would immediately abort decryption. This fix does not prevent all possible replays, given that iMessage accounts may be shared across multiple distinct devices. However, it would substantially reduce the impact of our attacks until a more permanent fix can be implemented. *Note: This modification has been incorporated into iOS 9.3 and Mac OS X*

10.11.4.

Force re-generation of all iMessage keys and destroy message logs. iMessage uses long-term decryption keys and offers no mechanism to provide forward secrecy. If possible, Apple should force all devices to re-generate their iMessage key pairs and destroy previously-held secret keys. In addition, Apple should destroy any archives of encrypted iMessage traffic currently held by the company.

Pin APSD/ESS certificates or sign ESS responses. The current iMessage protocol relies heavily on the security of TLS, both for communications with the key server and as an additional layer of protection for iMessage push traffic. Apple should enhance this security by employing certificate (or public key) pinning within the Messages application and apsd to prevent compromise of these connections. Alternatively, Apple could extend their proprietary signing mechanisms to authenticate key server responses as well as requests.

Reorganize message layout. The current layout of encrypted messages includes approximately 101 bytes of the CTR message within the RSA-OAEP ciphertext, which is resilient to ciphertext malleability attacks. Modifying sender-side code to re-organize the layout of the underlying plist data structure to incorporate the sender and receiver fields within this section of the message would immediately block our attack. Implementing this change requires two significant modifications: (1) Apple would need to disable dynamic construction of Huffman tables within the gzip compression, and (2) restructure the binary plist serialization code to place the sender address first. We stress that this is a fragile patch:

if any portion of the sender ID is left outside of the RSA ciphertext, the ciphertext again becomes vulnerable to mauling. Moreover, this fix will not protect group messages where the list of Recipients is longer than 100 bytes.

7.2 Long term recommendations

Replace the iMessage encryption mechanism. Apple should deprecate the existing iMessage protocol and replace it with a well-studied construction incorporating modern cryptographic primitives, forward secrecy and message authentication (e.g., OTR [17] or the TextSecure/Axolotl protocol [4]). At minimum, Apple should use a modern authenticated cipher mode such as AES-GCM for symmetric encryption. This change alone would eliminate our active attack on iMessage encryption, though it would still not address any weaknesses in the key distribution mechanism. In addition, iMessage should place the protocol versioning information within the public key block and the authenticated portions of the ciphertext, in order to prevent downgrade attacks.

Implement key transparency. While many of the protocol-level attacks described in this paper can be mitigated with protocol changes, iMessage’s dependence on a centralized key server represents an architectural weakness. Apple should take steps to harden iMessage against compromise of the ESS/IDS service, either through the use of key transparency [31] or by exposing key fingerprints to the user for manual verification.

8 Related Work

There are three lines of research related to our work: secure message protocols, attacks on symmetric encryption, and decryption attacks using compression schemes.

Instant messaging has received a great deal of attention from the research community. Borisov et al. introduced OTR [17] and proposed strong properties for messaging, such as per-message forward secrecy and deniability. Frosh et al. analyze descendant protocols such as TextSecure [24]. More recent work has focused on multi-party messaging [25] and improved key exchange deniability [39]. In a related area, Chen *et al.* analyzed push messaging integrations, including Apple push networking [20]. For a survey of secure messaging technologies, see [38].

A number of works have developed attacks on unauthenticated, or poorly authenticated, encryption protocols. In addition to the padding oracle of Vaudenay [40] and later applications [13], padding oracle attacks have been extended to use alternative side channels such as

timing [8, 19]. Some more recent works have proposed attacks on more complex data formats such as XML [27, 30].

Various works have addressed the combination of compression and encryption. Some attacks use knowledge of a relatively small number of bytes in the plaintext to learn information about the compression algorithm and eventually recover an encryption key [16, 37]. Kelsey [28] and others [29, 35] used compression in the (partially) chosen plaintext setting to recover information about plaintexts.

9 Conclusion

In this work we analyzed the security of a popular end-to-end encrypted messaging protocol. Our results help to shed light on the security of deployed messaging systems, and more generally, provide insight into the state of the art in security mechanisms currently deployed by industry. This insight raises questions about the way research results are disseminated and applied in industry and how our community should ensure that widely-used protocols employ best cryptographic practices.

This work leaves several open questions. First, the gzip format oracle attack we describe against iMessage may apply to other protocols as well. For example, OpenPGP encryption (as implemented by GnuPG) [18] also employs gzip and may be vulnerable to similar attacks when it is used for online applications such as instant messaging [2]. Moreover, our attack requires that the adversary have some access to a portion of the decrypted information. We leave to future work the development of a pure “blind” attack on gzip encryption, one that does not require this additional information.

10 Acknowledgments

This work was supported by: The National Science Foundation under awards CNS-1010928 and EFRI-1441209; The Office of Naval Research under contract N00014-14-1-0333; and The Mozilla Foundation.

We also are deeply grateful to Nate Cardozo and Kurt Opsahl of the Electronic Frontier Foundation for providing us with legal advice during our work on this paper.

References

- [1] iMessage. In OpenIM Wiki, Available at <https://imfreedom.org/wiki/IMessage>.
- [2] MCABBER. Available at <https://mcabber.com/>.
- [3] Pushproxy: A man-in-the-middle proxy for ios and os x device push connections. Available at <https://github.com/meeee/pushproxy>.

- [4] Textsecure. <https://github.com/WhisperSystems/TextSecure/wiki/ProtocolV2>. Accessed: 2014-11-13.
- [5] Trustwave to escape ‘death penalty’ for SSL skeleton key, 2012.
- [6] Apple pulls ad-blocking apps that can ‘compromise’ security. *Engadget* (October 2015).
- [7] ABELSON, H., ANDERSON, R., BELLOVIN, S. M., BENALOH, J., BLAZE, M., DIFFIE, W. W., GILMORE, J., GREEN, M., LANDAU, S., NEUMANN, P. G., RIVEST, R. L., SCHILLER, J. I., SCHNEIER, B., SPECTER, M. A., AND WEITZNER, D. J. Keys under doormats. *Commun. ACM* 58, 10 (Sept. 2015), 24–26.
- [8] ALFARDAN, N. J., AND PATERSON, K. G. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE S&P (Oakland) ’13* (2013), pp. 526–540.
- [9] APPLE COMPUTER. iOS Security: iOS 9.0 or later. Available at https://www.apple.com/business/docs/iOS_Security_Guide.pdf, September 2015.
- [10] APPLE INC. Privacy. Available at <http://www.apple.com/privacy/approach-to-privacy/>, 2015.
- [11] APUZZO, M., SANGER, D. E., AND SCHMIDT, M. S. Apple and other tech companies tangle with U.S. over data access. Available at <http://www.nytimes.com/2015/09/08/us/politics/apple-and-other-tech-companies-tangle-with-us-over-access-to-data.html>, September 2015.
- [12] BARBOSA, G. Apple execs Eddy Cue & Craig Federighi talk Apple Music, App Store & more in new interview. Available at <http://9to5mac.com/2016/02/12/apple-execss-eddy-cue-craig-federighi-talk-apple-music-app-store-more-in-new-interview/>, February 2016.
- [13] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., SIMIONATO, L., STEEL, G., AND TSAY, J.-K. Efficient padding oracle attacks on cryptographic hardware. In *CRYPTO ’12*, vol. 7417 of LNCS. Springer, 2012, pp. 608–625.
- [14] BELLARE, M., AND NAMPREMPRE, C. Authenticated encryption: relations among notions and analysis of the generic composition paradigm. *J. Cryptol.* 21, 4 (Sept. 2008), 469–491.
- [15] BELLARE, M., AND ROGAWAY, P. Optimal asymmetric encryption: How to encrypt with RSA. In *EUROCRYPT ’94* (1994), A. D. Santis, Ed., vol. 950 of LNCS. Springer, pp. 92–111.
- [16] BIHAM, E., AND KOCHER, P. C. A known plaintext attack on the PKZIP stream cipher. In *Fast Software Encryption: Second International Workshop, Leuven, Belgium, 14–16 December 1994, Proceedings* (1994), pp. 144–153.
- [17] BORISOV, N., GOLDBERG, I., AND BREWER, E. Off-the-record communication, or, why not to use PGP. *WPES ’04*, ACM Press, pp. 77–84.
- [18] CALLAS, J., DONNERHACKE, L., FINNEY, H., SHAW, D., AND THAYER, R. RFC 4880: OpenPGP Message Format. Available at <https://tools.ietf.org/html/rfc4880>, November 2007.
- [19] CANVEL, B., HILTGEN, A., VAUDENAY, S., AND VUAGNOUX, M. Password interception in a SSL/TLS channel. In *CRYPTO ’03*, vol. 2729 of LNCS. Springer Berlin Heidelberg, 2003, pp. 583–599.
- [20] CHEN, Y., LI, T., WANG, X., CHEN, K., AND HAN, X. Perplexed messengers from the cloud: Automated security analysis of push-messaging integrations. In *CCS ’15* (New York, NY, USA, 2015), CCS ’15, ACM, pp. 1260–1272.
- [21] CHIBA, D., MATSUDA, T., SCHULDT, J. C. N., AND MATSUURA, K. Efficient generic constructions of signcryption with insider security in the multi-user setting. In *ACNS ’11* (2011), pp. 220–237.
- [22] COVERT, A. Apple’s iMessage is the DEA’s worst nightmare. Available at <http://money.cnn.com/2013/04/07/technology/security/imessage-iphone-dea/>, April 2013.
- [23] DEUTSCH, P. RFC 1952: GZIP file format specification version 4.3, May 1996.
- [24] FROSCH, T., MAINKA, C., BADER, C., BERGSMA, F., SCHWENK, J., AND HOLZ, T. How secure is TextSecure? Cryptography ePrint Archive, October 2014.
- [25] GOLDBERG, I., USTAOĞLU, B., VAN GUNDY, M. D., AND CHEN, H. Multi-party off-the-record messaging. In *CCS ’09* (New York, NY, USA, 2009), CCS ’09, ACM, pp. 358–368.
- [26] GRIFFIN, A. WhatsApp and iMessage could be banned under new surveillance plans. *The Independent* (January 2015).
- [27] JAGER, T., AND SOMOROVSKY, J. How to break XML encryption. In *ACM CCS ’2011* (October 2011), ACM Press.
- [28] KELSEY, J. Compression and information leakage of plaintext. In *FSE ’02* (2002), vol. 2365 of LNCS, Springer, pp. 263–276.
- [29] KOHNO, T. Attacking and repairing the winZip encryption scheme. In *ACM CCS ’2004* (2004), ACM Press, pp. 72–81.
- [30] KUPSER, D., MAINKA, C., SCHWENK, J., AND SOMOROVSKY, J. How to break XML encryption – automatically. In *Proceedings of the 9th USENIX Conference on Offensive Technologies* (Berkeley, CA, USA, 2015), WOOT’15, USENIX Association.
- [31] MELARA, M. S., BLANKSTEIN, A., BONNEAU, J., FELTEN, E. W., AND FREEDMAN, M. J. CONIKS: Bringing key transparency to end users. In *USENIX ’15* (Washington, D.C., Aug. 2015), USENIX Association, pp. 383–398.
- [32] MESSIEH, N. Apple’s iMessage and Facetime blocked in the UAE. *TheNextWeb* (November 2011).
- [33] PALETTA, D. FBI Chief Punches Back on Encryption. *Wall Street Journal* (July 2015).
- [34] RAYNAL, F. iMessage privacy. Available at <http://blog.quarkslab.com/imessage-privacy.html>, October 2013.
- [35] RIZZO, J., AND DUONG, T. The CRIME Attack. Available at https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-1Ca2GizeuOfaLU2HOU/edit#slide=id.g1d134dff_1_222, September 2012.
- [36] SHIH, G., AND CARSTEN, P. Apple begins storing users’ personal data on servers in China. *Reuters* (August 2014).
- [37] STAY, M. ZIP attacks with reduced known plaintext. In *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2–4, 2001, Revised Papers* (2001), pp. 125–134.
- [38] UNGER, N., DECHAND, S., BONNEAU, J., FAHL, S., PERL, H., GOLDBERG, I., AND SMITH, M. SoK: Secure messaging. In *IEEE S&P (Oakland) ’15* (2015).
- [39] UNGER, N., AND GOLDBERG, I. Deniable key exchanges for secure messaging. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS ’15, ACM, pp. 1211–1223.
- [40] VAUDENAY, S. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS. In *EUROCRYPT ’02* (London, UK, 2002), vol. 2332 of LNCS. Springer-Verlag, pp. 534–546.
- [41] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.

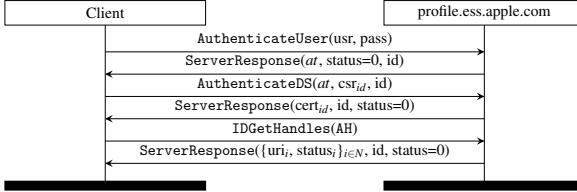


Figure 7: Profile conversation. usr = username, $pass$ = password, at = authentication token pt = push token, pk_{client} = client’s public key, st = session token. AH is an authentication header with the following fields: $cert_{device}$ = signed by the Apple Fairplay Certificate, $cert_{id}$ = a certificate associated with the client id, id , pt , nonce_{device}, nonce_{id}, σ_{device} , and σ_{id} .

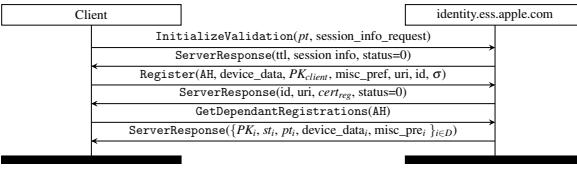


Figure 8: Identity conversation. pt = push token, pk_{client} = client’s public key, st = session token. AH is an authentication header with the following fields: $cert_{device}$ = signed by the Apple Fairplay Certificate, $cert_{id}$ = a certificate associated with the client id, id , pt , nonce_{device}, nonce_{id}, σ_{device} , and σ_{id} .

A Attacks on Key Registration

While this work focuses on the retrospective decryption of iMessage payloads, in the course of our reverse engineering we were able to implement attacks on Apple’s key registration infrastructure. The first attack is an implementation of attacks previously noted by Raynal *et al.* [34]. In these attacks, which work only against versions of iOS prior to iOS 9 and Mac devices prior to OS X 10.11.4 (i.e., devices without key pinning), an attacker with a forged Apple TLS certificate can intercept the connection to the Apple key server in order to substitute chosen public keys. Additionally, we find a novel attack against the device registration process that allows an attack with stolen credentials to circumvent existing protection mechanisms.

The protocol for registering a device is shown in Figure 8. The user first establishes a TLS connection to Apple’s IDS server and authenticates using their iCloud credentials. The client generates two separate key pairs: a 1280-bit RSA public key pair (pk_E, sk_E) for use in encrypting and decrypting messages and an ECDSA keypair (vk_S, sk_S) for authenticating messages. The client transmits the public portion of these keys $PK = (pk_E, vk_E)$ to the IDS, which registers it to the



Figure 9: Excerpts from an ESS/IDS directory lookup request (top) and response (bottom). The request address and a portion of the response Push token have been redacted.

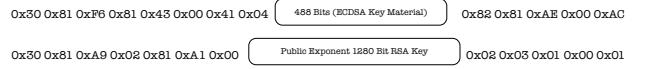


Figure 10: Format of public key payload in ESS server response

user’s iCloud account name. We diagram the full login and registration protocols in Figures 7 and 8. To support multiple devices on a single account, the IDS will store and return all public keys associated with a given account.

A.1 Key Substitution Attack

The Apple key distribution systems are accessed each time a legitimate user wants to send an iMessage to a new Recipient. The Messages client first contacts `query.ess.apple.com` to look up the keys for a given username. In response, the server returns the user’s public key(s), status, and push tokens for addressing APNs communications to the user. A fragment of the request and response is shown in Figure 9.

The `query.ess.apple.com` response message contains public keys, along with push tokens, for each of the devices registered to an account. Each of the key entries is a 332 character long base64 encoded binary payload. When decoded, they take the form shown in Figure 10.

Upon receiving the RSA public key in the above diagram, the Messages client uses this key to encrypt the outgoing iMessage payload. The ECDSA key is not used when sending a message, but is used to verify the integrity of a message when it is received from that user. iMessage clients appear to accept the most recent key delivered by ESS/IDS even if it disagrees with previous entries cached by the device.

Notably, the only security measures embedded in this conversation are authentication fields in the header of the *request*; the server does not sign the response. Thus the authenticity of the response depends entirely on the security of the TLS connection. This seems like an oversight, given that many other fields in the Apple protocols are explicitly authenticated. Worse, in iOS 8 and versions of OS X 10.11 released prior to December 2015, the Messages client does not use certificate pinning to ensure that the connection terminated at an Apple server. Thus an attacker with a stolen TLS root certificate can intercept key requests and substitute their own key as a response. This degrades the security of iMessage to that of TLS.

We implemented this attack by installing a self-signed X.509 root certificate into the local root certificate store of a Mac device. This allowed us to verify that there were no warning mechanisms that might alert a user to the key substitution. By further intercepting messages transmitted via the APNs network, we were able to respond to all key lookup requests with our own attacker key, and subsequently decrypt any iMessages transmitted via the device.

Our experiments demonstrate that iOS 9 is no longer subject to simple key substitution attacks, due to the addition of certificate pinning on TLS connections. This increases the relative impact of our novel decryption attacks. Surprisingly, our experiments demonstrated that OS X 10.11.1 remained vulnerable as of November 2015. We notified Apple of this oversight, and they have added key pinning as of OS X 10.11.13.

A.2 Credential theft

The first message in the registration process, shown in Figure 7, passes the user’s credentials to the `profile.ess.apple.com` server to be verified. As noted in previous sections, OS X 10.10.5 and iOS 8 devices do not employ certificate pinning on this server, and the credentials are sent in plaintext within the TLS connection.¹⁶ By conducting a TLS MITM attack on this connection, we are able to intercept iCloud login credentials. Using this information we can register new iMessage devices to an account, ensuring that we will be able

¹⁶OS X 10.11 devices do not employ certificate pinning on this connection either, but they do not appear to send the credentials in plaintext.

to receive future messages.

Apple’s primary defense against registration of new devices is a notification message that is sent to all previously-registered devices. In order to register a new device to a target account without alerting the victim, we also developed a method to overcome these notification mechanisms. We observed two such mechanisms:

1. Upon registration of a new device, all devices logged into the account receive a push notification over the APNs network. In response, each device initiates the `GetDependantRegistrations` call shown in Figure 8.
2. When an iMessage account is registered to a device that has not previously been registered to that account, a notification email is generated and sent to the account’s registered email.

In the first instance, once the APNs push notification signaling that a `GetDependantRegistrations` call should be executed has arrived at a client, the client will continuously send the request until it receives a response. An active attacker on the victim’s network can simply block all these requests, but this is not sustainable over long periods of time. We discovered that the client is satisfied when it receives any response — even a poorly formatted unreadable one. Thus, an attacker can edit the server response causing it to decode incorrectly. The client will accept this response and terminate the repeated `GetDependantRegistrations` calls. This blocks notifications that would alert the victim to the fact that a new device has been registered to their account. All subsequent iMessage traffic, both incoming and outgoing, will be forwarded to the attack device. Until a user logs out of their iMessage client, logs into a new iMessage client, or manually checks the list of devices associated with their account, they will never notice that their traffic is being forwarded to the attack device.

A.3 Updates in OS X 10.11

The ESS messaging protocol changed in a number of ways with the 10.11 update to OS X. The exchange of credentials for an authorization token has moved to point to `gsa.apple.com` and that connection has certificate pinning implemented. Due to this fact, we are unable to MITM this connection, but attempting to login to an account with bad credentials will result only in a message to that server and an error message displayed on the client. Additionally, there is a message sent to `setup.icloud.com` with a username and password pair in which the password is no longer transmitted in plaintext.

The key substitution attack still worked against OS X 10.11 versions as of November 2015, but the additional certificate pinning of apsd made it more difficult to intercept the message. In order to make sure the attack still functioned properly, we recovered the encrypted payload of the message from the apsd logs and were able to successfully decrypt the message using our own keys. Although we are not able to easily intercept the messages as we could with 10.10.5, this attack still effectively reduces the security of iMessage to that of TLS.

B Bypassing TLS

To execute the attacks described in this paper, the attacker must obtain encrypted iMessages from the APNs link. Since iMessage secures the APNs connection using TLS, this requires the attacker to penetrate to the TLS encryption on the link between Apple and the end-device.

We identified three approaches to bypassing TLS on the APNs connections: (1) Apple, or an attacker with access to Apple’s infrastructure, can intercept the contents of push messages as they transit the APNs servers; (2) on certain iOS and OS X versions that do not include certificate pinning for APNs, an attacker with access to a stolen CA root certificate may be able to conduct an MITM attack on the TLS connection; or (3) on the same versions, an attacker can “sideload” a root certificate on the target device, by briefly taking physical control of it, or convincing a victim to install a root certificate via a malicious email or web page. The latter technique is particularly concerning due to the similarity between Apple’s interface for installing root CAs and other non-critical certificate installation requests that may be presented to the user (see Figure 11). Since some Apple operating systems do not use certificate pinning, installation of a root certificate allows arbitrary interception of both APNs and HTTPS connections.

We identified attacks (2) and (3) as infeasible on all iOS 9 versions due to the inclusion of certificate pinning on APNs connections in that operating system. As of November 2015 when we first notified Apple of the results in this paper, we discovered that the then-current version of OS X 10.11 did not include certificate pinning. In response to our disclosure, Apple added certificate pinning to OS X as of December 2015.

We stress that given the interest in iMessage expressed by nation-states [26], a compromise of CA infrastructure cannot be ruled out. Even without such attacks, there have been several recent examples of CA-signed root or intermediate certificates being issued for use within corporate middle-boxes, primarily for the purposes of enterprise TLS interception [5]. TLS interception may occur even within Apple OS distributions: a recent incident involving iOS 9 allowed ad-blocking software to install a

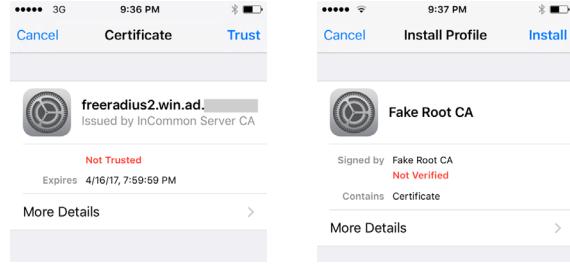


Figure 11: On the left is a certificate verification dialog presented on encountering an unknown wireless access point. On the right is a root CA installation dialog.

TLS root certificate [6].

Predicting, Decrypting, and Abusing WPA2/802.11 Group Keys

Mathy Vanhoef

iMinds-DistriNet, KU Leuven

Mathy.Vanhoef@cs.kuleuven.be

Frank Piessens

iMinds-DistriNet, KU Leuven

Frank.Piessens@cs.kuleuven.be

Abstract

We analyze the generation and management of 802.11 group keys. These keys protect broadcast and multicast Wi-Fi traffic. We discovered several issues and illustrate their importance by decrypting all group (and unicast) traffic of a typical Wi-Fi network.

First we argue that the 802.11 random number generator is flawed by design, and provides an insufficient amount of entropy. This is confirmed by predicting randomly generated group keys on several platforms. We then examine whether group keys are securely transmitted to clients. Here we discover a downgrade attack that forces usage of RC4 to encrypt the group key when transmitted in the 4-way handshake. The per-message RC4 key is the concatenation of a public 16-byte initialization vector with a secret 16-byte key, and the first 256 keystream bytes are dropped. We study this peculiar usage of RC4, and find that capturing 2^{31} handshakes can be sufficient to recover (i.e., decrypt) a 128-bit group key. We also examine whether group traffic is properly isolated from unicast traffic. We find that this is not the case, and show that the group key can be used to inject and decrypt unicast traffic. Finally, we propose and study a new random number generator tailored for 802.11 platforms.

1 Introduction

In the last decennia, Wi-Fi became a de facto standard for medium-range wireless communications. Not only is it widely supported, several new enhancements also make it increasingly more performant. One downside is that (encrypted) traffic can easily be intercepted. As a result, securing Wi-Fi traffic has received considerable attention from the research community. For example, they showed that WEP is utterly broken [11, 42, 4], demonstrated attacks against WPA-TKIP [43, 45, 47, 41], performed security analysis of AES-CCMP [24, 39, 13], studied the security of the 4-way handshake [17, 18, 34], and so on.

However, most research only focuses on the security of pairwise keys and unicast traffic. Group keys and group traffic have been given less attention, if mentioned at all.

In this paper we show that generating and managing group keys is a critical, but underappreciated part, of a modern Wi-Fi network. In particular we investigate the generation of group keys, their transmission to clients, and the isolation between group and unicast traffic. We discovered issues during all these phases of a group key's lifetime. To address some of our findings, we propose and implement a novel random number generator that extracts randomness from the physical Wi-Fi channel.

First we study the random number generator proposed by the 802.11 standard. Among other things, the Access Point (AP) uses it to generate group keys. Surprisingly, we find that it is flawed by design. We argue that implementing the algorithm as specified, results in an unacceptably slow algorithm. This argument is supported empirically: all implementations we examined, modified the generator to increase its speed. We demonstrate that these modified implementations can be broken by predicting the generated group key within mere minutes.

The generated group keys are transferred to clients during the 4-way WPA2 handshake. We found that it is possible to perform a (type of) downgrade attack against the 4-way handshake, causing RC4 to be used to encrypt the transmission of the group key. We analyze the construction of the per-message RC4 key and its effect on biases in the keystream. This reveals that an attacker can abuse biases to recover an 128-bit group key by capturing 2^{30} to 2^{32} encryptions of the group key, where the precise number depends on the configuration of the network.

Group keys should only be used to protect broadcast or multicast frames. In other words, pairwise and group keys should be properly isolated, and unicast packets should never be encrypted with a group key. An AP can enforce this by only sending, but never receiving, group addressed frames. However, all APs we tested did not provide this isolation. We demonstrate that this allows

FC	addr1	addr2	addr3	KeyID / PN	Data
----	-------	-------	-------	------------	------

Figure 1: Simplified 802.11 frame with a WPA2 header.

an attacker to use the group key to inject, and in turn decrypt, any traffic sent in a Wi-Fi network.

Finally, we propose and study a novel random number generation tailored for 802.11 platforms. It extracts randomness from the wireless channel by collecting fine-grained Received Signal Strength Indicator (RSSI) measurements. These measurements can be made using commodity devices even if there is no background traffic. We show our algorithm can generate more than 3000 bits per second, and even when an adversary can predict individual RSSI measurements with high probability, the output of the generator still remains close to uniformly random.

To summarize, our main contributions are:

- We show that the 802.11 random number generator is flawed, and break several implementations by predicting its output, and hence also the group key.
- We present a downgrade-style attack against the 4-way handshake, allowing one to recover the group key by exploiting weaknesses in the RC4 cipher.
- We show that the group key can be used to inject and decrypt any (internet) traffic in a Wi-Fi network.
- We propose and study a random number generator that extracts randomness from the wireless channel.

The rest of this paper is organized as follows. Section 2 introduces relevant parts of the 802.11 standard. We break the random number generator of 802.11 in Section 3. Section 4 presents a downgrade attack against the 4-way handshake, and attacks on its usage of RC4. In Section 5 we use the group key to inject and decrypt any frames, including unicast ones. In Section 6 we propose a new random number generator. Finally, we explore related work in Section 7, and conclude in Section 8.

2 Background

This section provides a background on the 802.11 protocol, the 4-way handshake, and the RC4 stream cipher.

2.1 The 802.11 Protocol

When a station wishes to transmit data, it needs to add a valid 802.11 header (see Figure 1). This header contains the necessary MAC addresses to route the frame:

```
addr1 = Receiver MAC address
addr2 = Sender MAC address
addr3 = Destination MAC address
```

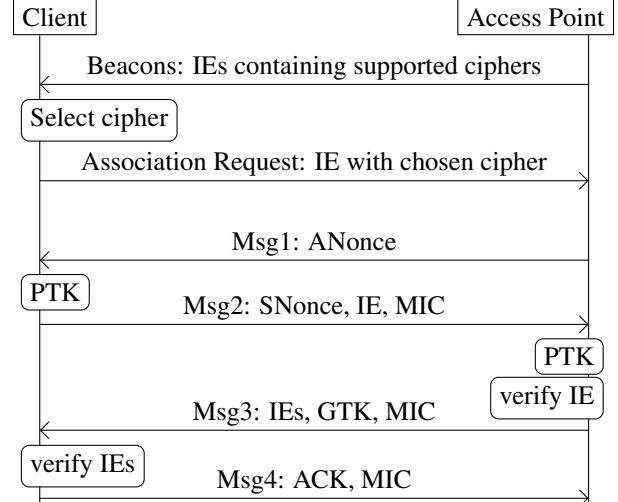


Figure 2: Discovering APs by listening to beacons, followed by the association and 4-way WPA2 handshake.

The Access Point (AP) forwards received frames to their destination, which is either a node on the wired network, or a Wi-Fi client. In frames received by a client, `addr1` should equal `addr3`, hence no further routing is required. For example, when a client sends an outbound IP packet, `addr1` equals the address of the AP, `addr2` contains his own address, and `addr3` equals the address of the router.

If a client wishes to transmit a broadcast or multicast frame, i.e., a group addressed frame, he first sends it as a unicast frame to the AP. This means `addr1` equals the address of the AP, and `addr3` equals the broadcast or multicast destination address. The AP then encrypts the frame using the group key if needed, and broadcasts it to all associated clients. This assures all clients within the range of the AP will receive the frame, even if certain stations are not within range of each other.

The Frame Control (FC) field contains, among other things, the ToDS and FromDS flags. The ToDS flag is set if the frame is sent from a client to an AP, and the FromDS flag is set if the frame is sent in the reverse direction. The fifth field in Figure 1 is only included when encryption is used, and contains the Key ID and Packet Number (PN). The PN prevents replay attacks. The 2-bit Key ID field is only used in group addressed frames, where it identifies which group key is used to protect and encrypt the frame.

2.2 Discovering APs and Negotiating Keys

Clients can discover APs by listening for beacons, which are periodically broadcasted by the AP (see Figure 2). These beacons contain the supported cipher suites of the AP in Information Elements (IEs). When a client wants to connect to an AP, and has selected a cipher to use, it starts by sending an association request to the AP. This

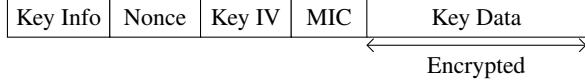


Figure 3: Simplified layout of EAPOL-Key frames.

request includes the selected cipher in an information element (IE). To prevent downgrade attacks, the client and AP will verify the received and selected IEs in the 4-way handshake. In this handshake, the client and AP also authenticate each other, and negotiate a Pairwise Temporal Key (PTK). The PTK is essentially the set of negotiated session keys. The first part of the PTK is called the Key Confirmation Key (KCK), and is used to authenticate handshake messages. The second part is called the Key Encryption Key (KEK), and is used to encrypt any sensitive data in the handshake messages. Finally, the third part is called the Temporal Key (TK), and is used to protect data frames that are transmitted after the handshake. To assure a new PTK is generated, both the client and AP first generate unpredictable, random nonces called SNonce and ANonce, respectively. The PTK is then derived from a shared secret or passphrase, the ANonce and SNonce, and the MAC addresses of the client and AP.

In the first message of the 4-way handshake, the AP sends the ANonce to the client (see Figure 2). On receipt of this message, the client calculates the PTK. In the second message, the client sends the SNonce, the IE representing the previously selected cipher, and includes a Message Integrity Code (MIC) calculated over the complete message. Note that the MIC is calculated using the KCK key contained in the PTK. After receiving Msg2 and the SNonce, the AP also derives the PTK. At this point both parties know the PTK, and all messages are authenticated using a MIC. Using the KCK and received MIC, the AP verifies the integrity of Msg2. The AP then checks whether the included IE matches the IE that was received in the initial association request. If these IEs differ, the handshake is aborted. Otherwise the AP replies with the Group Temporal Key (GTK), and its supported ciphers as a list of IEs. The client verifies the integrity of Msg3, and compares the included IEs with the ones previously received in the beacons. If the IEs differ, the handshake is aborted. Otherwise the client finishes the handshake by sending Msg4 to the AP.

Messages in the 4-way handshake are defined using EAPOL-Key frames, whose most important fields are shown in Figure 3. The Key Info field contains flags identifying which message this frame represents in the handshake. It also states which algorithm is used to calculate the MIC, and which cipher is used to encrypt the Key Data field (see Section 4). Note that the KCK key is used to calculate the MIC, and that the KEK key is used to encrypt the Key Data field. Finally, the Key IV

Key Scheduling (KSA)	Keystream Output (PRGA)
<pre>L = len(key) j, S = 0, range(256) for i in range(256): j += S[i] + key[i % L] swap(S[i], S[j]) return S</pre>	<pre>S, i, j = KSA(key), 0, 0 while True: i += 1 j += S[i] swap(S[i], S[j]) yield S[S[i] + S[j]]</pre>

Figure 4: Implementation of RC4 in Python-like pseudo-code. All additions are carried out modulo 256.

field may contain an initialization vector (IV) to assure the Key Data field is always encrypted using a unique key. The most common usage of the Key Data field is to transport the group key (GTK), and to transfer any IEs.

2.3 The RC4 Stream Cipher

RC4 is a fast and well-known stream cipher consisting of two algorithms: a Key Scheduling Algorithm (KSA) and a Pseudo-Random Generation Algorithm (PRGA). Both are shown in Figure 4. The KSA takes as input a variable-length key, and generates a permutation S of the set $\{0, \dots, 255\}$. This gradually changing permutation, combined with a public counter i and a private index j , form the internal state of the PRGA. In each algorithm, a swap operation is performed near the end of every round. We use the notations i_t , j_t , and S_t , for the indices i and j and the permutation S after round t . Rounds are indexed based on the value of i after the swap operation. Hence the KSA has rounds $t = 0, \dots, 255$ and the PRGA has rounds $t = 1, 2, \dots$. We let Z_r denote the keystream byte outputted at round r . Whenever it might not be clear whether we are referring to the KSA or PRGA, we use the notations S_t^{KSA} and S_t^{PRGA} , respectively.

Multiple biases have been found in the first few keystream bytes of RC4. These are called short-term biases. Arguably the most well known was found by Mantin and Shamir [30]. They showed that the value zero occurs twice as often at position 2 compared to uniform. In contrast, there are also biases that keep occurring throughout the whole keystream. We call these long-term biases. For example, Fluhrer and McGrew (FM) found that the probability of certain consecutive bytes deviate from uniform throughout the whole keystream [12]. Similarly, Mantin discovered a long-term bias towards the pattern $ABSAB$, where A and B represent byte values, and S a short sequence of bytes called the gap [29]. Letting g denote the length of the gap, the bias can be written as follows:

$$\Pr[Z_r, Z_{r+1} = Z_{r+g+2}, Z_{r+g+3}] = 2^{-16} \left(1 + \frac{e^{(-4-8g)/256}}{256} \right)$$

Hence the longer the gap, the weaker the bias.

Listing 1: Random number generator as proposed by the 802.11 standard in Python-like pseudocode [21, §M.5].

```

1 def PRF-256(key, label, data):
2     R = HMAC-SHA1(key, label + "\x00" + data + "\x00")
3     R += HMAC-SHA1(key, label + "\x00" + data + "\x01")
4     return R[:32]
5
6 def Hash(data):
7     return PRF-256(0, "Init Counter", data)
8
9 def NetworkJitter():
10    if ethernet traffic available:
11        return LSB(receive time of ethernet packet)
12    else:
13        Start 4-way handshake, stop after receiving Msg2
14        return LSB(Msg1.sent_time) + LSB(Msg2.rssi)
15        + LSB(Msg2.receive_time) + Msg2.snonce
16
17 def GenRandom():
18    local = "\x00" * 32
19    # Wait for Ethernet traffic or association, and
20    # loop until result is "random enough" or 32 times
21    for i in range(32):
22        buf = Hash(macaddr + currttime + local + i)
23        for j in range(32):
24            local += NetworkJitter()
25    return Hash(macaddr + currttime + local + "\x20")

```

3 Breaking the 802.11 RNG

In this section we argue that the Random Number Generator (RNG) of 802.11 is flawed, and break several implementations by predicting the generated group key.

3.1 The Proposed RNG in 802.11

The security enhancements amendment to 802.11, called 802.11i, includes a software-based RNG [22, §H.5.2]. It extracts randomness from clock jitter and frame arrival times. While the standard states the proposed algorithm is only expository, and real implementations should extend it with other sources of entropy, we found that several platforms directly implement it and even simplify it.

Listing 1 contains the proposed RNG as the function `GenRandom`. The outer for-loop first calculates a hash over the MAC address of the station, the current time, the `local` variable, and the loop counter. Then it makes multiple calls to `NetworkJitter` in order to collect randomness from the arrival times of Ethernet or Wi-Fi frames. Here `LSB` returns the least significant byte of a timestamp. Note the comment on line 20, which is copied almost verbatim from the standard. It instructs to either run the outer loop 32 times, or until the `lcoal` variable is “random enough”. No clarification is made on what this exactly means. The standard also mentions that the variable `currttime` can be set to zero if the current time is not available. However, there is no discussion on how

this impacts the RNG, e.g., whether additional iterations of the outer for-loop should be executed. Additionally, the standard does not mandate a minimum resolution for the timestamps that are used. It only states that the send and receive timestamps of frames should use the highest resolution possible, *preferably* 1 ms or better. Finally, the RNG is executed on demand, i.e., there is no state saved between two invocations of `GenRandom`.

3.2 Analysis

A careful inspection of the RNG shows it is ill-defined and likely insecure. One problem lies with the `NetworkJitter` function, which is called 256 times by `GenRandom`. First, the if test on line 10 is ambiguous. If it checks whether there was Ethernet traffic in the last x seconds, repeated calls will probably operate on the same Ethernet packet, and the function will return the same data. On the other hand, if this test implies monitoring the Ethernet interface for x seconds, this might cause a total delay of $256 \cdot x$ seconds. Furthermore, the value of x is not given. In any case, either calling `NetworkJitter` implies waiting a significant amount of time until there is new traffic, or repeated calls return the same value.

When the second clause of the if statement on line 10 is taken, the arrival times of frames transmitted during the 4-way handshake are used. Specifically, it mentions to initiate the 4-way handshake. This is something only an AP can do when a client is trying to connect to this AP (see Figure 2). Therefore the proposed algorithm is only usable by APs. We also remark that if the AP were to constantly abort the handshake after receiving message 2 (see line 13), most clients will blacklist the AP for a certain period. During this period, the client will no longer attempt to connect to the network. Hence it becomes infeasible to initiate and abort 256 4-way handshakes, which is something the random number generator is supposed to do. We conclude that the function `NetworkJitter` is unusable in practice. Based on this we conjecture, and empirically confirm in Section 3.4, that vendors will not implement this function.

Another design flaw is that no state is kept between subsequent calls to `GenRandom`. Hence its output depends only on a small amount of network traffic and timestamp samples. A better design is to collect randomness in a pool, and regularly reseed this pool with new randomness. When done properly, this protects against permanent compromise of the RNG, iterative guessing attacks, backtracking attacks, and so on [25, 3, 9].

We conclude that the proposed RNG is questionable at best. Either it returns bytes having a low amount of entropy, or calling it will incur significant slowdowns. In Section 3.4 we will show that in practice this construction results in defective and predictable RNGs.

3.3 Generation of the Group Key

The 802.11 standard defines, but does not mandate, a key hierarchy for the generation of group keys [21, §11.6.1]. This hierarchy is described in Listing 2, where `macaddr` denotes the MAC address of the AP, and `currtime` is either the current time or zero. The `on_startup` function is executed at boot time, and generates a random auxiliary key called the Group Master Key (GMK). Additionally, it initializes the `key_counter` variable to a pseudo-random value [21, §11.6.5]. Actual group keys, called Group Temporal Keys (GTKs), are derived from the GMK and `key_counter` using a Pseudo-Random Function (PRF) in `new_gtk`. The length of the generated GTK depends on the cipher being used to protect group traffic. If TKIP is used, the GTK is 32 bytes long. If CCMP is used, the GTK is 16 bytes long. This implies the PRF has to generate either 128 or 256 bits of keying material, depending on the configuration of the network. Hence we use the function name `PRF-X`, where the value of X depends on the amount of requested keying material. Note that the implementation of `PRF-256` is shown in Listing 1, and that `PRF-128` closely resembles this function. The latest standard also states [21, §11.6.1.4]:

“The GMK is an auxiliary key that may be used to derive a GTK at a time interval configured into the AP to reduce the exposure of data if the GMK is compromised.”

However, this makes no sense: there is no point in introducing a new key to reduce the impact if that key itself leaks. Curiously, we found that older versions of the standard did not contain this description of the GMK. Instead, older versions stated that the GMK may be reinitialized to reduce the exposure of data in case the current value of the GMK is ever leaked.

Most implementations renew the GTK every hour by calling a function similar to `new_gtk`. More importantly, the `key_counter` variable is also used to initialize the Key IV field of certain EAPOL-Key frames (see Figure 3). After using the value of `key_counter` for this purpose, it is incremented by one. Since these IVs are public values, the value of `key_counter` is known by adversaries. We found that some implementations even use `key_counter` to generate nonce values during the 4-way handshake, though this is not recommended as it may enable precomputation attacks [21, §8.5.3.7].

One major disadvantage of the proposed key hierarchy, is that fresh entropy is never introduced when generating a new group key in `new_gtk`. Hence, once the value of GMK has been leaked, or recovered by an attacker, all subsequent groups keys can be trivially predicted.

Since the standard assumes that `GenRandom` provides cryptographic-quality random numbers, there appears

Listing 2: Python-like pseudocode describing the group key hierarchy (and generation) according to the 802.11 standard.

```
1 def on_startup():
2     GMK, key = GenRandom(), GenRandom()
3     buf = macaddr + currtime
4     key_counter = PRF-256(key, "Init Counter", buf)
5
6 def new_gtk():
7     gnonce = key_counter++
8     buf = macaddr + gnonce
9     GTK = PRF-X(GMK, "Group key expansion", buf)
```

to be no advantage in using this key hierarchy. Instead, the AP can directly call `GenRandom` to generate new group keys. Some consider this key hierarchy a relic from older 802.11 standards, which did not yet require that devices must implement a strong RNG [20]. Perhaps the only (unintended) advantage this construction has, is that the *first* group key is now determined by two calls to `GenRandom`, instead of only one call. Hence, if an adversary is trying to attack a weak implementation of `GenRandom`, he has to predict its output twice (see Section 3.4). Nowadays implementations are allowed to directly generate a random value for the GTK [21, §11.6.1.4], though many platforms still implement the proposed group key hierarchy (see Section 3.4).

3.4 Practical Consequences

We now study the RNG of real 802.11 platforms. First we focus on popular consumer devices. To estimate the popularity of a specific brand, we surveyed wireless networks in two Belgian municipalities. We were able to recognize specific brands based on vendor-specific information elements in beacons. We detected 6803 networks, and found that MediaTek- and Broadcom-based APs alone covered at least 22% of all Wi-Fi networks. We will focus on both because of their popularity. Additionally we examine Hostapd for Linux. Finally, we study embedded systems by analysing the Open Firmware project. We found that, apart from Hostapd, all these platforms produce predictable random numbers.

3.4.1 MediaTek-based Routers

Access points with a MediaTek radio use out-of-tree Linux drivers to control the radio¹. These drivers directly manage the 4-way handshake and key generation. They implement the 802.11 RNG as shown in Listing 1, but do not call `NetworkJitter`. This strengthens our hypothesis that this function is infeasible to implement in

¹Available from www MEDIATEK.com/en/downloads

practice. It also means the only source of randomness is the current time, for which it uses the *jiffies* counter of the Linux kernel. This counter is initialized to a fixed value at boot, and incremented at every timer interrupt. The number of timer interrupts per second is configured at compile time and commonly lies between 100 and 1000. Hence it is a coarse grained timestamp, meaning the `currtime` variable likely has the same value each time it is sampled in `GenRandom`. That is, the current time is the only random source being used, and provides little entropy.

The group key hierarchy is implemented according to the 802.11 standard, with one exception. Instead of initializing `gnonce` to `key_counter` in line 7 of Listing 2, it generates a new value using `GenRandom`, and assigns the result to `gnonce`.

We show that this RNG is flawed by predicting the group key generated by an Asus RT-AC51U. A similar approach can be followed for other routes that also use a MediaTek radio. The first step is to predict the GMK generated at boot. By recompiling the firmware, and printing out the jiffies values that were used at the start and end of an invocation of `GenRandom`, we observe that it uses at most two different values. Note that we printed these values out only after calling `GenRandom`, to assure we did not noticeably influence the used jiffies values. Hence the jiffies values is incremented at most by one while executing `GenRandom`. Since this increment may happen in any of the 32 loops, `GenRandom` can result in total 32 possible values if the initial jiffies value is known. If AES-CCMP is used to protect group traffic, the initial jiffies value when generating the GMK lies in the range $[2^{32} - 72\,889, 2^{32} - 72\,884]$. If WPA-TKIP is used, it lies in $[2^{32} - 73\,067, 2^{32} - 73\,061]$. The number of attached USB or Ethernet devices, amount of Ethernet traffic, or other Wi-Fi options, did not impact these estimates. Since it is trivial to determine whether AES or TKIP is used, and less than 10 possible initial values are used in both cases, we end up with at most $32 \cdot 10$ possible values for the GMK.

The second step is to estimate the jiffies count when the GTK, i.e., group key, was generated. By default, a new group key is generated every hour. Hence, if we know the uptime of the router, we can determine when the current group key was generated. Conveniently, beacons leak the uptime of a device in their timestamp field. This field is used to synchronize timers between all stations [21, §10.1], and is generally initialized to zero at boot. Hence its value corresponds to the uptime of the router. From this we can estimate the jiffies counter's value at the time the group key was generated. However, as the device keeps running, clock skew will affect our prediction. By logging jiffies values, we observed that the clock skew over one month made our prediction off

by at most 4500 jiffies. Therefore, even after an uptime of year, our prediction of the jiffies value will only be off by roughly 50000. In other words, we conjecture that the prediction after a year will be off by at most 200 seconds.

We created an OpenCL program to search for the group key on a GPU. It tests candidate keys by decrypting the first 8 bytes of a packet, and checking if they match the predictable LLC/SNAP header. If the targeted router has been running for one year, it has to test $320 \cdot 50\,000 \cdot 32 \approx 2^{29}$ candidates to recover the group key. However, testing each key is rather costly, as it involves calculating $33 \cdot 4$ SHA-1 hashes to derive the group key, and we must then decrypt the first 8 bytes of the packet to verify the key. Nevertheless, on our NVIDIA GeForce GTX 950M, it takes roughly two minutes to test all 2^{29} candidates and recover the GTK. We confirmed this by successfully predicting several group keys generated by our Asus RT-AC51U, when it had an uptime of more than a month. We conclude the group key generated by a MediaTek driver can be brute-forced using commodity hardware in negligible time.

3.4.2 Broadcom Network Authentication

The network access server of Broadcom implements the 4-way handshake, including the necessary key generation. It implements the group key hierarchy according to the 802.11 standard (see Listing 7). Additionally, it uses the `key_counter` variable to initialize the Key IV field of EAPOL-Key frames. However, it does not implement the RNG as proposed in the 802.11 standard. Instead, the RNG it uses depends on the kernel used by the device.

When running on a VxWorks or eCos kernel, random numbers are generated by taking the MD5 checksum of the current time in microsecond accuracy. Hence random numbers are straightforward to predict. And since the output of MD5 is used, only 16 bytes of supposedly random data is generated in every call. One widely used device that uses a VxWorks kernel, is version 5 or higher of the popular WRT54G router. Furthermore, the Apple AirPort Extreme also uses a VxWorks kernel. To predict the group key generated by these devices, we only have to predict the value of GMK. Recall that the value of `key_counter` is leaked in the Key IV field of certain EAPOL-Key fields, and can simply be passively sniffed. Since GMK is a 32-byte value, it is initialized by calling the random number generator twice. In order to predict the output of these two calls, we must first determine the time at which the group key was generated based on the uptime of the router. Similar to the MediaTek case, the uptime can be derived from the timestamp field in beacons. Assuming we can estimate time at which the group key was generated with an accuracy of one second, and that the timestamp in the next call to the RNG differs by

Listing 3: Generation of random nonces by the Open Firmware project in Python-like pseudocode.

```
1 def on_system_boot():
2     rn = lcg_next(milliseconds since boot)
3     data = macaddr + rn
4     rn = lcg_next(rn)
5     nonce = PRF-256(rn, "Init Counter", data)
6
7 def lcg_next(rn):
8     return rn * 0x107465 + 0x234567
9
10 def compute_next_snonce():
11     nonce += 1
12     return nonce
```

at most 10 ms, we have to test $1000000 \cdot 10000 \approx 2^{33}$ keys. We implemented an OpenCL program to simulate this search, and on our GeForce GTX 950M, it takes around 4 minutes to test all candidate keys. Hence the generated group keys by this RNG can be predicted using commodity hardware.

When running on a Linux kernel, random bytes are read from `/dev/urandom`. This is problematic since, on routers and embedded devices, `/dev/urandom` is commonly predictable at boot [19]. And since entropy for the group keys is only collected at boot (see Section 3.3), this again means all groups keys may be predictable.

3.4.3 The Linux Hostapd Daemon

Hostapd implements the 802.11 group key hierarchy as shown in Listing 2. However, when generating a new group key using a function similar to `new_gtk`, it also samples and incorporates new entropy. Additionally, Hostapd does not implement the 802.11 RNG. Instead, it generates keys by reading from `/dev/random`. In case insufficient entropy is available, it will re-sample from `/dev/random` when the first client is attempting to connect. In case there still is not enough entropy available, the client is not allowed to connect. All combined, this means the keys used by Hostapd should be secure.

3.4.4 Open Firmware (OpenBoot)

The Open Firmware project, previously called Open-Boot, is a free and open source boot loader programmed in Forth². Most notably it is used in the One Laptop Per Child project. Interestingly, it provides basic but secure Wi-Fi functionality during the early stages of the boot process. Since at this stage no operating system is loaded, we consider it an ideal candidate to investigate how vendors implement RNGs in a constrained (embedded) environment.

²Available from [svn://openbios.org/openfirmware](http://openbios.org/openfirmware)

Currently, the Wi-Fi module of Open Firmware only provides client functionality. Therefore we focus on the generation of random nonces during the 4-way handshake. Listing 3 illustrates how Open Firmware generates these nonces. Summarized, when loading the Wi-Fi module, a random initial nonce is generated, and this nonce is incremented whenever it is used. In this regard, the algorithm follows the 802.11 standard [21, §11.6.5]. However, the generation of the initial random nonce is very weak. It takes the uptime of the device in number of milliseconds, runs this twice through a linear congruential generator, combines it with its own MAC address, and finally expands this data using a Pseudo-Random Function (PRF). All this information can be predicted or brute-forced by an adversary.

We attribute this weak construction to a careless implementation, and treat it as an indication that a better design is to let the Wi-Fi chip generate random numbers itself. Users can then query the Wi-Fi chip when new randomness is needed. In Section 3 we demonstrate how a strong random number generator can be implemented using commodity Wi-Fi devices.

4 RC4 in the 4-Way Handshake

In this section we present a (type of) downgrade attack against the 4-way handshake. As a result, RC4 is used to encrypt sensitive information in the handshake. We present two attacks against the usage of RC4 in the handshake, and show how it allows an attacker to recover the group key. We also determine the performance of our attacks, and propose countermeasures.

4.1 Downgrading to RC4

When inspecting the 4-way handshake in Figure 2, we can see that the AP sends the group key (GTK) to the client before the client verifies the IEs of the AP. Recall that these IEs contain the supported cipher suites of the AP, which are advertised in plaintext beacons. In other words, the client can only detect that a downgrade attack has occurred *after* the AP has transmitted the group key in Msg3. This is problematic because, if multiple ciphers can be used to protect the handshake, an adversary can try to perform a downgrade attack to induce the AP into encrypting and transmitting the group key using a weak cipher.

Interestingly, the 4-way handshake can indeed be protected by several cipher suites [21, §11.6.2], meaning a downgrade attack is possible. More specifically, the cipher suite that is used to protect the handshake is determined by two settings that may, or may not, be requested by the client in its association request (see Figure 2). In

Table 1: Cipher suites used in the 4-way handshake.

Selected Options	Ciphers Used
Fast Transition (FT)	AES-CMAC, AES key wrap
CCMP without FT	HMAC-SHA1, AES key wrap
TKIP without FT	HMAC-MD5, RC4

particular, when support for fast network transitions is requested, AES-CMAC and NIST AES key wrap are used to protect messages in the 4-way handshake. Otherwise, the cipher used to protect the handshake depends on the pairwise cipher that will be used to protect normal data frames transmitted after the handshake. In case CCMP will be used, the 4-way handshake uses HMAC-SHA1 and NIST EAS key wrap. More troublesome, if TKIP will be used, then HMAC-MD5 and RC4 are used to authenticate and encrypt data, respectively. An overview of this selection process is shown in Table 1.

Our idea is now to create a rogue AP that only advertises support for TKIP. Hence victims wanting to connect to the AP will use TKIP, and in turn the group key transmitted in the 4-way handshake will be encrypted using RC4. This works because the client will only detect the downgrade attack after receiving message 3. However, to make the AP send message 3, it must first receive and successfully verify the integrity of message 2. If its integrity cannot be verified, which is done using the negotiated session keys, the AP will not continue the handshake. Since the session keys depend on the MAC addresses of the client and AP, it means we must create a rogue AP with the same MAC address as the real one. Fortunately this is possible by performing a channel-based man-in-the-middle attack [46]. Essentially, the attacker clones the AP on a different channel, and forwards packets to, and from, the real AP. The MAC addresses in forwarded frames are not modified. This assures the station and AP will generate the same session keys, meaning the AP will successfully verify the authenticity of message 2. This man-in-the-middle position allows the attacker to reliably manipulate messages. In particular, it will use this position to modify the beacons and probe responses so it seems the AP only supports (WPA-)TKIP. Hence the client will be forced to select TKIP, causing the AP use to RC4 for encrypting the group key.

We tested this downgrade-style attack against a network that advertised both support for TKIP and CCMP. Since the rogue AP only advertised support for TKIP, the victim indeed selected TKIP. We then confirmed that the AP encrypts the group key using RC4, and that the client detected our attack only after receiving message 3.

Interestingly, the 4-way handshake uses RC4 in a rather peculiar manner [22, §8.5.2j]. The per-message RC4 key is the concatenation of the 16-byte Initial-

ization Vector (IV) and the 16-byte Key Encryption Key (KEK). Additionally, the initial 256 keystream bytes are dropped. This construction is similar to the one used by WEP, except that the IV is longer, and that some initial keystream bytes are dropped. Interestingly, using a longer IV likely weakens this per-message key construction [28, 36], while dropping the initial keystream bytes should strengthen it [33, 28]. In the next two sections, we analyze the impact of this peculiar combination.

4.2 Recovering the Key Encryption Key

We first examine whether it is possible to perform a key recovery attack similar to those that broke WEP [11, 42]. In general, these attacks are applicable if a public IV is prepended (or appended) to a fixed secret key. This matches the construction of the per-message key K in the 4-way handshake, where the public 16-byte IV is prepended to the secret but static 16-byte KEK key. More formally, the per-message key is constructed as follows:

$$K = IV \parallel KEK$$

Although the first 256 keystream bytes of RC4 are dropped, Mantin showed this does not prevent key recovery attacks [28]. We will adapt Mantin’s attack to the 4-way handshake, and study whether it is feasible to perform this attack in practice.

Similar to the original FMS attack [11], Mantin’s extension of the FMS attack uses an iterative process to recover the key K [28]. That is, each iteration assumes the first x bytes of K are known, and attempts to recover the next key byte $K[x]$. In the first step of each iteration of Mantin’s new attack, keystreams are collected that were generated with an IV for which the condition $S_{x-1}^{KSA}[1] = x$ holds. We call these *applicable* IVs. Mantin proved that applicable IVs leak information about the key byte $K[x]$ through the following relation [28]:

$$\Pr[K[x] = S_{x-1}^{-1}[i_{257} - z_{257}] - j_{x-1} - S_{x-1}[x]] \approx 1.1 \cdot 2^{-8} \quad (1)$$

This relation can be used to recover $K[x]$ with a simple voting mechanism as follows. Each applicable IV casts a vote for a certain value through equation (1). After all IVs are processed, the value with the most votes is assumed to be the value of $K[x]$. Unfortunately, this has the downside that a single incorrect guess for any byte means the complete key K is also wrong. To mitigate this, we pick the C most likely values for each byte, and construct C^{16} candidate values for K . Each candidate can be tested based on the captured IVs and corresponding keystreams. We simulated this attack against the 4-way handshake, with as goal to determine how many applicable IVs have to be captured to recover the KEK key. The result of this simulation is shown in Figure 5. To obtain

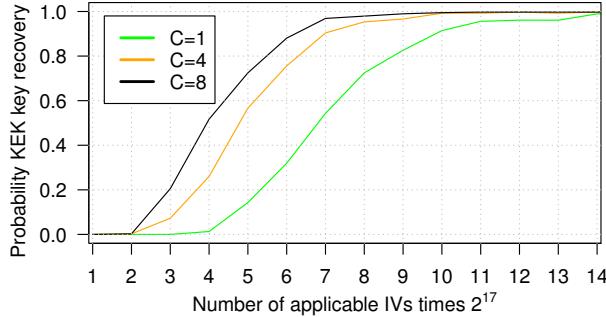


Figure 5: Probability of finding the 16-byte KEK key given the number of applicable IVs and the branch factor C .

a 80% success ratio of recovering the *KEK*, roughly 2^{21} applicable IVs have to be collected. Note that to execute the attack, we must be able to determine the value of z_{257} . Fortunately, this is possible by relying on the predictable IEs that are located at the start of the EAPOL Key Data field, meaning we can derive the keystream at these initial positions.

We now determine how much effort it takes to collect the required number of applicable IVs. First notice that the most likely way the condition $S_{x-1}^{KSA}[1] = x$ is satisfied, is when the value x is swapped into position 1 at round 1 of the KSA (recall Listing 4). Or more formally, that $j_1 = x$, since $K[x]$ is likely not modified in the first round. Indeed, the 15 other rounds only affect position 1 if j ever equals 1. Assuming a random initial IV is used, this will not happen with a probability of $(\frac{255}{256})^{15} = 0.94$. At round one, $j_1 = K[0] + 1 + K[1]$ (we assume $K[0] \neq 1$ which holds with high probability). Hence, with high probability, an IV is applicable when $K[0] + K[1] = x - 1$.

The 802.11 standard states that a station must generate an initial random IV at startup, and increment this IV after it is used in a message [21, §11.6.5]. In other words, the IV is used as a counter. However, it does not specify whether little or big endian counters must be used. Our experiments indicate that most devices we use the IV as a big endian counter. Fortunately, from a defenders perspective, this means that generating the required number of applicable IVs takes an enormous amount of time. Assuming that the condition $K[0] + K[1] = x - 1$ does not hold, we must wait until $K[1]$ has been incremented sufficiently many times. However, only every $256^{14} = 2^{112}$ IVs does the value of $K[1]$ change. Clearly, this means that collecting the required number of IVs is infeasible when the AP uses a big endian counter. If the IV is generated by a little endian counter, the condition $K[0] + K[1] = x - 1$ is generally satisfied every 256th message. This means around $256 \cdot n$ messages must be

collected in order to have roughly n applicable IVs for all iterations.

While it is possible to generate many handshakes by forcibly disconnecting clients, new handshakes will use a different KEK key. Since our attack assumes that the KEK is constant, this is not an option. The only method we identified to make the AP send several handshake messages protected by the same KEK, is by not acknowledges them, and letting the AP retransmit them. Note that each retransmission uses a new IV. Unfortunately, only a few messages will be retransmitted before the AP gives up and aborts the handshake process. In the next Section we present an attack that does tolerate frequent changes of the KEK key. We conclude that the 4-way handshake, as defined in the 802.11i standard, is vulnerable to key recovery attacks. However, these attacks seem difficult to pull off against popular implementations.

4.3 Plaintext Recovery Attacks

We now turn our attention to plaintext recovery attacks, where an adversary targets information that is repeatedly encrypted under different RC4 keys. Previous work on RC4 has shown that these types of attacks can be very successful, with attacks against TLS and TKIP being on the verge of practicality [2, 47]. In particular, the attack against WPA-TKIP by Paterson et al. [37] is fairly similar to our scenario. They showed that for WPA-TKIP, the public 3-byte prefix of the per-message RC4 key induces large, prefix-dependent, biases into the RC4 keystream [37, 15]. An adversary can precompute these prefix-dependent biases, and mount a powerful plaintext recovery attack against the first few bytes encrypted by RC4. This inspired us to investigate whether the public 16-byte IV used in the 4-way handshake also induces IV-dependent biases, even though the first 256 keystream bytes are dropped. Hence we examine the biases induced by the public IV contained in EAPOL frames, and then demonstrate through simulations that these can be used to recover the group key.

It is impossible to empirically investigate every possible IV, since this would mean inspecting 2^{128} values. Instead, we initially generated detailed keystream statistics for four randomly selected IVs. This indicated that large, IV-dependent biases indeed persist in the keystream, even after the first 256 bytes (which are dropped). Motivated by this result, we took the all-zero initialization vector IV_0 , and investigated how changing the values at each specific position in the IV influences the keystream distribution. Changing a byte at position x to value y is denoted by $IV_0[x] = y$. The generation of all datasets took more than 13 CPU years.

Figure 6a and 6b show the keystream distribution for the initialization vectors IV_a and IV_b , respectively. The

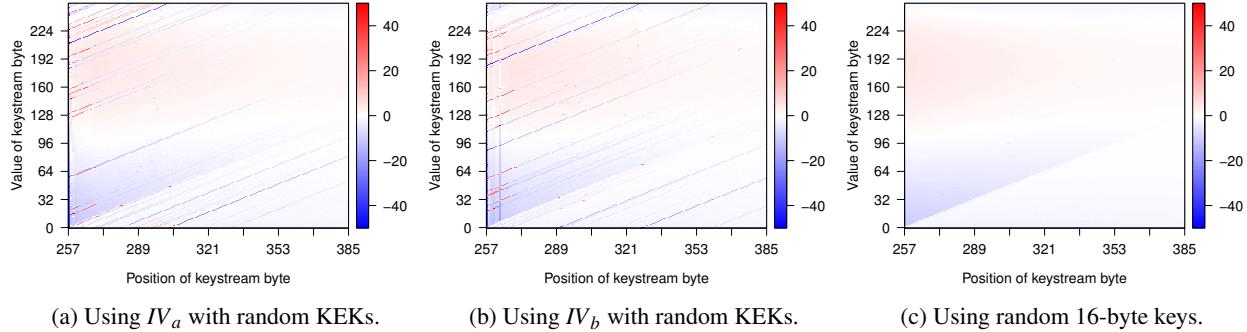


Figure 6: Biases in the RC4 keystream when concatenating a fixed 16-byte IV with a random 16-byte key (here called KEK key), and when using random 16-byte keys. Each points encodes a bias as the number $(pr - 2^{-8}) \cdot 2^{24}$, capped to values in $[-50, 50]$, with pr the empirical probability of the keystream byte value (y-axis) at a given location (x-axis).

distributions of IV_c and IV_d behave similarly. Their values were randomly generated in Python and are:

$$\begin{aligned} IV_a &= 0x2fe931f824ef842bf262dbca357bb31c \\ IV_b &= 0x48d9859f9fa08bb1599744a20491dd49 \\ IV_c &= 0x6c1924761b03faf8decc0dfc09dd3078 \\ IV_d &= 0xe31257489cbe7d91e5365286c26f5023 \end{aligned}$$

These keystream distributions were generated using 2^{45} RC4 keys for each IV. For comparison, Figure 6c and Figure 9c shows the keystream distribution for fully random 16-byte keys, generated using roughly 2^{47} RC4 keystreams [47]. We observe that the initialization vector induces strong biases that are visible as straight lines, even after position 256. By comparing these biases with the keystream distribution of 16-byte random RC4 keys in Figure 6c, we can conclude that the biases represented by the light and red background, are not caused by the specific IV values. Instead, they appear inherent to RC4.

We also generated 16 datasets, where in each dataset the value at one specific position in the all-zeros IV is changed. That is, for $0 \leq x \leq 15$, we generated datasets for the vectors $IV_0[x] = y$, where y ranges between 0 and 255. Each dataset was generated using 2^{43} keys, resulting in rather noisy distributions. Nevertheless, an inspection of these datasets confirmed that each IV induces specific biases, visible as straight lines in our graphs. A more detailed discussion of these biases is out of scope, and is left as future work.

We now use the IV-dependent biases to recover repeated plaintext, in order to get an indication of how well a plaintext recovery attack works against the 4-way handshake. This is done by combining the precomputed keystream distributions with captured ciphertexts, in order to calculate likelihood estimates for each plaintext value. The actual plaintext value is then assumed to be the one with the highest likelihood. Since our goal is mainly to evaluate the performance of the resulting at-

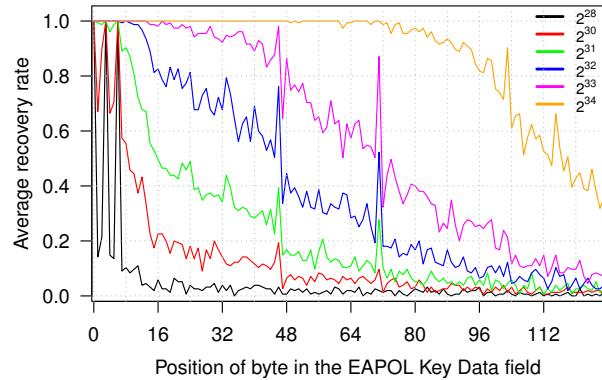


Figure 7: Success rate of decrypting a byte in the EAPOL Key Data field, in function of the byte position and number of collected ciphertexts. The legend shows the total number of ciphertexts used, where half of the ciphertexts were generated using IV_a , and the other half using IV_b .

tacks, we refer to previous work for the technicalities behind these calculations [2, 37, 47]. In particular, we implemented the binning algorithm proposed by Patterson et al. [37], and the single-byte candidate generation algorithm proposed by Vanhoef and Piessens [47]. To keep the computations feasible, we assumed that half of the captured ciphertext were generated by IV_a , and the other half by IV_b . For the binning algorithm of Patterson et al., Figure 7 shows the probability of correctly decrypting a byte. For the candidate generation algorithm, which returns a list of plaintext candidates for a sequence of bytes, we first need to determine at which position the group key is stored in the EAPOL Key Data field.

The location of the group key depends on which cipher suites are supported. If only TKIP is supported in a RSN network, it starts at position 30. In contrast, if both TKIP and AES are supported, and if the older WPA informa-

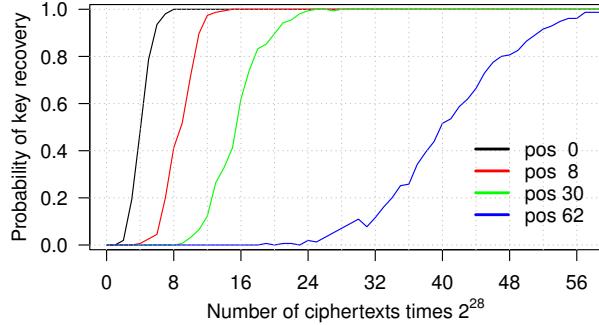


Figure 8: Probability of recovering a 16-byte key using short-term biases and 2^{26} candidates. The legend shows its starting position in the EAPOL key data field.

tion elements are included in addition to the RSN IEs³, the group key starts at position 62. For other configurations, the group key is located somewhere between position 30 and 62. The probability of recovering a 16-byte key at these positions, in function of the number of captured ciphertexts (handshakes), is shown in Figure 8. We remark that if the group key starts at position 62, attacks that exploit Mantin’s ABSAB bias become more efficient [29, 6]. This is because the 62 bytes that precede the group key are predictable, and hence an attack similar to the one that broke RC4 in TLS can be launched [47].

Finally, in principle it is also possible to attack group key update messages, since the 802.11 standard does not mandate that these messages should be encrypted using the pairwise cipher [21, §11.6.7]. Group key updates use EAPOL frames, and are sent when the AP generates a new group key. Interestingly, in these messages the group key is either located at position 8, or at position 0. The success rate of recovering a 16-byte key at these positions is shown in Figure 8. For example, if the key starts at position 8, roughly 2^{31} encryptions of the GTK have to be captured in order to decrypt it. Since in this case there is little surrounding known plaintext, it is the best attack an adversary can launch [2, 47, 6].

4.4 Countermeasures

To prevent the downgrade attack, APs should disable support of WPA-TKIP. Even when an adversary creates a rogue AP advertising TKIP, the real AP will reject any request for TKIP, and hence will never use RC4 in the 4-way handshake. Similarly, clients should not connect to a network using WPA-TKIP.

³Early implementations based on the draft 802.11i standard use WPA IEs, instead of RSN IEs as used in modern networks.

5 Abusing the Group Key

In this section we show that the group key can be used to decrypt and inject any traffic, including unicast traffic.

5.1 Injecting Unicast Frames

First we explain how to inject unicast traffic using the group key. This is not possible by simply encrypting a unicast frame with the group key, and setting the KeyID field to the id used by the group key. The receiver always uses pairwise keys to decrypt unicast frames, and ignores the KeyID value (recall Section 2.1). Furthermore, it is not possible to encapsulate unicast IP packets in group addressed frames. Indeed, RFC 1122 states that stations should discard unicast IP packets that were received on a broadcast or multicast link-layer address [5, §3.3.6]. However, this check is only performed when the packet is passed on to the IP layer. Since an AP does not operate at the IP layer, but on the MAC layer, it does not perform this check. Inspired by this observation, we encrypt the unicast IP packet using the group key, and send it to the AP. For the three address fields in the frame we use the following values:

```
addr1 = FF:FF:FF:FF:FF:FF
addr2 = Spoofed sender MAC address
addr3 = Spoofed destination MAC address
```

Although in a normal network the AP never processes group addressed frames, we found that APs can be forced to process our injected broadcast packet by setting the ToDS bit in the Frame Control (FC) field. To assure the correct value of the KeyID field is used, an adversary can monitor other broadcast frames, or brute-force this value. The AP will then decrypt this packet using the group key. It will notice that the destination address (addr3) does not equal its own MAC address, and hence will forward the frame to the actual destination. If the destination MAC address is another wireless station, the AP will encrypt the frame using the appropriate pairwise key, and transmit it. As a result, the receiver will decrypt and process the forwarded, now unicast, frame. If the destination address is not a wireless station, it will be forwarded over the appropriate Ethernet connection. This technique can be used to inject IP packets, ARP packets, and so on, using the group key.

5.2 Decrypting All Traffic

Since unicast frames are encrypted with pairwise keys, we cannot directly use the group key to decrypt them. Nevertheless, it is possible to trick stations into sending all IP traffic to the broadcast MAC address, meaning the

group key will now be used to encrypt this traffic. This is done by performing an ARP poisoning attack. The malicious ARP packets are injected using the technique presented in Section 5.1. In our attack, we poison the ARP cache of the client so the IP address of the gateway is associated with the broadcast MAC address. Similarly, on the router, the IP address of the client will also be associated with a broadcast address. Since IP addresses in local networks are generally predictable, an attacker can brute-force the IP address of the client and router. After the ARP poisoning attack, all IP packets sent by the client and router are encrypted using the group key. An attacker can now capture and decrypt these packets. Furthermore, he can forward them to their real destination using the (unicast) packet injection technique of Section 5.1, so the victim will not notice he is under attack.

5.3 Experimental Verification

We tested this attack against an Asus RT-AC51U and a laptop running Windows 7. The group key was obtained by exploiting the weak random number generator as discussed in Section 3.4.1. In order to successfully perform the ARP poisoning attack against Windows, we injected malicious ARP requests. First, we were able to successfully inject the ARP packets using the group key. This confirms that the group key can be used to inject unicast packets. Once we poisoned the ARP cache of both the victim and router, they transmitted all their packets towards the broadcast MAC address. At this point we were able to successfully decrypt these broadcast packets using the group key, and read out the unicast IP packets sent by both the victim and router.

5.4 Countermeasures

If the network is operating in infrastructure mode, the AP should ignore all frames with a broadcast or multicast receiver address. This prevents an attacker from abusing the AP to forward unicast frames to stations. Another option is to disable all group traffic. While this may seem drastic, it is useful for protected but public hotspots. In these environments, connected stations do not trust each other, meaning group keys should not be used at all. Interestingly, the upcoming Hotspot 2.0 standard already supports this feature under the Downstream Group Addressed Forwarding (DGAF) option [49]. If DGAF is disabled, no group keys are configured, meaning the stations and AP ignore all group addressed Wi-Fi frames.

6 A New RNG for 802.11 Platforms

In this section we propose a random number generator that extracts randomness from fine-grained Received

Signal Strength Indicator (RSSI) values. Specifically, we rely on the spectral scan feature of commodity 802.11 radios. This gives us roughly three million RSSI measurements per second, even if there is no background traffic.

6.1 Spectral Scan Feature

Most Atheros Wi-Fi radios, such as the AR9280, can perform RSSI measurements over the 56 sub-carriers used in high throughput (HT) OFDM⁴. Atheros calls this feature a spectral scan. It matches the requirement to decode HT OFDM modulated frames, where the channel is divided into 64 subcarriers [21, §18]. Eight of these subcarriers are used as guards to avoid channel cross talk, and are thus not sampled, resulting in 56 usable subcarriers. Therefore the spectral scan feature matches the normal OFDM demodulation requirements, and should be straightforward to implement by other vendors as well. The sweep time of one sample, i.e., spectral scan, is 4μs, and these scans can be made even when there is no background traffic. Each RSSI measurement is reported as an 8-bit value. After some optimizations, we could make our AR2980 chip generate around 50k samples per second. Since each sample contains 56 RSSI values, this totals to roughly three million measurements per second.

6.2 Random Number Generation

In our random number generator, we want to extract randomness out of every single RSSI measurement. Since our commodity devices can generate a large number of measurements per second, even when there is no background traffic, we need a fast method to process all these measurements. Hence our main goal is to design a technique to rapidly process all measurements. The resulting output can then be given as input to a system that properly extracts and manages randomness (see for example Yarrow-160 [25], or the model by Barak and Halevi [3] and its improvements [9]). In other words, our goal is only to design a method to rapidly process RSSI measurements which can be implemented in Wi-Fi radios, and to assess the quality of the resulting output.

We start by deriving one (possibly biased) bit out of each RSSI measurement. Any biases will be suppressed in a later step. Due to random variations in the background noise, the transmissions of other stations, and internal imperfections of the hardware, antenna, and radio, we expect that each RSSI measurement contains some amount of randomness. More concretely, we expect that the least significant bit of each RSSI measurement displays the most amount of randomness. To also take into account the other bits, we perform an exclusive or over all bits in the 8-bit RSSI measurement. Even if the other

⁴See <http://wikidevi.com/wiki/Atheros> for a list.

Table 2: Average number statistical test results for various configurations of the random number generator.

Configuration	Pass	Poor	Weak	Fail
1 bit per subcarrier	97.2%	0%	2.8%	0%
1 bit per spectral scan	98.1%	0%	1.9%	0%
normal mode	98.1%	0%	1.9%	0%

bits are not random, this can only increase the overall randomness. Since we are extracting 1 bit per subcarrier, we call this initial generator the “1 bit per subcarrier” configuration. When running the Dieharder statistical test suite [7] on this configuration, we noticed promising results (see Table 2). The Dieharder suite is a reimplementation of the Diehard tests [31], and in addition contains several tests from the NIST test suite [40]. While none of the tests fail, on average 2.8% of the tests return a weak result. However, this only means that the generated bits do not contain any obvious deficiencies. Subtle or small biases may still be present, and have to be filtered out. We do this by relying on the large number of measurements that our commodity devices can generate.

To suppress possible biases in the 1-bit per subcarrier construction, we combine several bits using an exclusive-or chain. More formally, if we have a sequence of bits b_1, b_2, \dots, b_n , the exclusive-or chain of these bits is $bit = b_1 \oplus b_2 \oplus \dots \oplus b_n$. Assuming that each bit b_i is equal to one with probability p , combining n bits in this manner has the following characteristics [10]:

$$\Pr[bit = 1] = 0.5 - 2^{n-1} \cdot (p - 0.5)^n \quad (2)$$

$$\Pr[bit = 0] = 0.5 + 2^{n-1} \cdot (p - 0.5)^n \quad (3)$$

We can now see that as n goes to infinity, both probabilities approach 0.5, meaning any possible biases will be suppressed. Moreover, an exclusive-or chain should also be straightforward to implement in a Wi-Fi radio.

In the second version of our generator, we use the exclusive-or chain to generate one random bit for each spectral scan sample. That is, all 56 random bits extracted from the subcarriers are XOR’ed together. We call this the “1-bit per spectral scan” mode. The reasoning behind this construction is that one bit is now influenced by all subcarriers, i.e., all available frequencies. An attacker that wants to influence the generation of any bit, now has to predict or influence all 56 subcarriers. The new results of the Dieharder tests show this improves the quality of the random numbers (see Table 2).

In a last step we combine 16 bits generated using the 1-bit per spectral scan construction. Again this is done using an exclusive-or chain. We call this the “normal mode”. Interestingly, the results of the Dieharder test suite no longer improve. We conjecture that the 1.9% of

tests that are marked as weak are statistical flukes: even a random stream of bits can look non-random at times.

Finally, we remark that in the normal execution mode of the generator, in total 56 · 16 bits are XOR’ed together. Hence, even if an attacker can correctly predict the 1 bit per subcarrier with a probability of 98%, our normal execution mode still outputs one bit that is close to uniform. More precisely, by relying on equation 2 and 3, the returned bit equals one with a probability of approximately $0.5 \cdot (1 - 2^{-52})$. Hence, by relying on the large number of measurements returned by the radio chip, even a very powerful attacker is unlikely to predict the final output of the generator. Furthermore, these bits are outputted at a speed of roughly 3125 bits per second. Finally, we believe that our technique can be efficiently implemented in Wi-Fi chips themselves. In practice, implementations can then query the Wi-Fi chip for random samples, and properly and securely manage this collected randomness using a model such as the one proposed by Barak and Halevi [3], or one of its improvements [9].

7 Related Work

While the random number generators that are used in certain browsers [14], OpenSSL [8], Linux [3, 16], GNU Privacy Guard [35], FreeBSD [50], and so on, have been widely studied, we are not aware of any works that study the random number generator of 802.11. More closely related to our work, Lorente et al. discovered that many routers generate weak, and sometimes predictable, default WPA2 passwords [27]. However, the random number generator of 802.11 is not used for this purpose, and hence was not analyzed.

The security of the 4-way handshake has been studied in several works [17, 18, 34]. These works revealed denial-of-service vulnerabilities [17, 34], or proof of the security of an improved design [18]. Additionally, they focus on whether an attacker can perform a downgrade attack against the cipher used to protect traffic transmitted after the handshake. In contrast, we study downgrade attacks against the ciphers used to protect the handshake itself. The 802.11 standard also contains an informative analysis of the handshake [21, 11.6.6.8].

Many researchers have studied RC4 and its usage. Key recovery attacks against WEP were discovered [11], and were later improved in other works [48, 44, 43, 42]. In particular, Mantin and Klein studied whether the WEP key can still be recovered if the initial 256 bytes of RC4 are dropped [28, 26]. We extend this analysis by studying the impact of 16-byte initialization vectors as used in the 4-way handshake, and perform simulations of resulting attacks. AlFardan et al. showed that the initial 256 bytes of RC4 are biased [2]. Vanhoef and Piessens extended this result and showed that bytes between position 256

and 512 are also biased. [47]. In [6] Bricout et al. analyze the structure and exploitation of Mantin’s *ABSAB* bias.

Security of group keys, and the isolation between unicast and group traffic, is briefly mentioned in the Hole 196 vulnerability [1]. However, this attack assumes that an associated (trusted) client will abuse the group key. Therefore it can only be considered an insider threat. Furthermore, it does not discuss how to inject unicast traffic using the group key, nor does it show how all internet traffic can be decrypted using the group key.

Several previous works use the RSSI measurements of 802.11 frames, as returned by commodity Wi-Fi radios, to create secret key agreement protocols [32, 23, 38]. Such a protocol negotiates a shared secret between two stations, that is unpredictable by observers. These works use the average RSSI over all subcarriers, meaning some entropy is lost compared to our per-subcarrier measurements. We use the spectral scan feature to perform RSSI measurements, which makes it possible to generate these measurements even if there is no background traffic. Models to properly collect and manage randomness, such as those contained in RSSI measurements, have also been studied. Examples are Yarrow-160 [25], the model by Barak and Halevi [3], or the model by Dodis et al. [9].

8 Conclusion

Although the generation of pairwise 802.11 keys has been widely analyzed, we have shown the same is not true for group keys. For certain devices the group key is easily predictable, which is caused by the faulty random number generator proposed in the 802.11 standard. This is especially problematic for Wi-Fi stacks in embedded devices, as they generally do not have other (standardized) sources of randomness. Furthermore, we have demonstrated a downgrade attack against the 4-way handshake, resulting in the usage of RC4 to protect the group key. An adversary can abuse this in an attempt to recover the group key.

We also showed that the group key can be used to inject any type of packet, and can even be used decrypt all internet traffic in a network. Combined with the faulty 802.11 random number generator, this enables an adversary to easily bypass both WPA-TKIP and AES-CCMP. To mitigate some of these issues, we also proposed and implemented a strong random number generator tailored for 802.11 platforms.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven. Mathy Vanhoef holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO).

References

- [1] AHMAD, M. S. Wpa too! In *DEF CON* (2010).
- [2] ALFARDAN, N. J., BERNSTEIN, D. J., PATERSON, K. G., POTTERING, B., AND SCHULDT, J. C. N. On the security of RC4 in TLS and WPA. In *USENIX Security* (2013).
- [3] BARAK, B., AND HALEVI, S. A model and architecture for pseudo-random generation with applications to /dev/random. In *CCS* (2005).
- [4] BITTAU, A., HANDLEY, M., AND LACKEY, J. The final nail in WEP’s coffin. In *IEEE SP* (2006).
- [5] BRADEN, R. Requirements for internet hosts – communication layers. RFC 1122, 1989.
- [6] BRICOUT, R., MURPHY, S., PATERSON, K. G., AND VAN DER MERWE, T. Analysing and exploiting the mantin biases in RC4. *Cryptology ePrint Archive*, Report 2016/063, 2016.
- [7] BROWN, R. G. Dieharder: A random number test suite. Available from <http://www.phy.duke.edu/~rgb/General/dieharder.php>, Feb. 2016.
- [8] CHECKOWAY, S., NIEDERHAGEN, R., EVERSPAUGH, A., GREEN, M., LANGE, T., RISTENPART, T., BERNSTEIN, D. J., MASKIEWICZ, J., SHACHAM, H., AND FREDRIKSON, M. On the practical exploitability of Dual EC in TLS implementations. In *USENIX Security* (2014).
- [9] DODIS, Y., POINTCHEVAL, D., RUHALT, S., VERGNIAUD, D., AND WICHES, D. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 647–658.
- [10] FAIRFIELD, R., MORTENSON, R., AND COULTHART, K. An LSI random number generator. In *CRYPTO* (1984).
- [11] FLUHRER, S., MANTIN, I., AND SHAMIR, A. Weaknesses in the key scheduling algorithm of RC4. In *SAC* (2001).
- [12] FLUHRER, S. R., AND McGREW, D. A. Statistical analysis of the alleged RC4 keystream generator. In *FSE* (2000).
- [13] FOUCHE, P.-A., MARTINET, G., VALETTE, F., AND ZIMMER, S. On the security of the CCM encryption mode and of a slight variant. In *Applied Cryptography and Network Security* (2008).
- [14] GOLDBERG, I., AND WAGNER, D. Randomness and the netscape browser. *Dr. Dobb’s Journal* (1996).
- [15] GUPTA, S. S., MAITRA, S., MEIER, W., PAUL, G., AND SARKAR, S. Dependence in IV-related bytes of RC4 key enhances vulnerabilities in WPA. *Cryptology ePrint Archive*, Report 2013/476, 2013.
- [16] GUTTERMAN, Z., PINKAS, B., AND REINMAN, T. Analysis of the linux random number generator. In *IEEE SP* (2006).
- [17] HE, C., AND MITCHELL, J. C. Analysis of the 802.1 i 4-Way handshake. In *WiSe* (2004), ACM.
- [18] HE, C., SUNDARARAJAN, M., DATTA, A., DEREK, A., AND MITCHELL, J. C. A modular correctness proof of IEEE 802.11i and TLS. In *CCS* (2005).
- [19] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security* (2012).
- [20] IEEE. Motions to address some letter ballot 52 comments. In *802.11 WLANs WG proceedings* (2003).
- [21] IEEE STD 802.11-2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, 2012.
- [22] IEEE STD 802.11i. *Amendment 6: Medium Access Control (MAC) Security Enhancements*, 2004.

- [23] JANA, S., PREMNATH, S., CLARK, M., KASERA, S., PATWARI, N., AND KRISHNAMURTHY, S. On the effectiveness of secret key extraction from wireless signal strength. In *MobiCom* (2009).
- [24] JONSSON, J. On the security of CTR+ CBC-MAC. In *SAC* (2002).
- [25] KELSEY, J., SCHNEIER, B., AND FERGUSON, N. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography* (1999), Springer, pp. 13–33.
- [26] KLEIN, A. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography* (2008).
- [27] LORENTE, E. N., MEIJER, C., AND VERDULT, R. Scrutinizing WPA2 password generating algorithms in wireless routers. In *USENIX WOOT* (2015).
- [28] MANTIN, I. A practical attack on the fixed RC4 in the WEP mode. In *AsiaCrypt* (2005).
- [29] MANTIN, I. Predicting and distinguishing attacks on RC4 key-stream generator. In *EUROCRYPT* (2005).
- [30] MANTIN, I., AND SHAMIR, A. A practical attack on broadcast RC4. In *FSE* (2001).
- [31] MARSAGLIA, G. Diehard tests of randomness. <http://stat.fsu.edu/pub/diehard/>.
- [32] MATHUR, S., TRAPPE, W., MANDAYAM, N., YE, C., AND REZNIK, A. Radio-telepathy: extracting a secret key from an unauthenticated wireless channel. In *MobiCom* (2008).
- [33] MIRONOV, I. (Not so) random shuffles of RC4. In *CRYPTO* (2002).
- [34] MITCHELL, C. H. J. C. Security analysis and improvements for IEEE 802.11i. In *NDSS* (2005).
- [35] NGUYEN, P. Q. Can we trust cryptographic software? cryptographic flaws in GNU privacy guard v1.2.3. In *EUROCRYPT* (2004).
- [36] PATERSON, K. G., POETTERING, B., AND SCHULDT, J. C. Big bias hunting in amazonia: Large-scale computation and exploitation of RC4 biases. In *AsiaCrypt* (2014).
- [37] PATERSON, K. G., SCHULDT, J. C. N., AND POETTERING, B. Plaintext recovery attacks against WPA/TKIP. In *FSE* (2014).
- [38] PATWARI, N., CROFT, J., JANA, S., AND KASERA, S. High-rate uncorrelated bit extraction for shared secret key generation from channel measurements. *TMC* (2010).
- [39] ROGAWAY, P., AND WAGNER, D. A critique of CCM. Cryptology ePrint Archive, Report 2003/070, 2003.
- [40] RUKHIN, A., SOTO, J., NECHVATAL, J., SMID, M., AND BARKER, E. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Tech. rep., DTIC Document, 2001.
- [41] SEPEHRDAD, P., SUSIL, P., VAUDENAY, S., AND VUAGNOUX, M. Tornado attack on RC4 with applications to WEP & WPA. Cryptology ePrint Archive, Report 2015/254, 2015.
- [42] STUBBLEFIELD, A., IOANNIDIS, J., AND RUBIN, A. D. A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *TISSEC* (2004).
- [43] TEWS, E., AND BECK, M. Practical attacks against WEP and WPA. In *WiSec* (2009).
- [44] TEWS, E., WEINMANN, R.-P., AND PYSHKIN, A. Breaking 104 bit WEP in less than 60 seconds. In *JISA*. 2007.
- [45] VANHOEF, M., AND PIJESSENS, F. Practical verification of WPA-TKIP vulnerabilities. In *ASIA CCS* (2013), ACM, pp. 427–436.
- [46] VANHOEF, M., AND PIJESSENS, F. Advanced Wi-Fi attacks using commodity hardware. In *ACSAC* (2014).
- [47] VANHOEF, M., AND PIJESSENS, F. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security* (2015).
- [48] VAUDENAY, S., AND VUAGNOUX, M. Passive-only key recovery attacks on RC4. In *SAC* (2007).
- [49] WI-FI ALLIANCE. *Hotspot 2.0 (Release 2) Technical Specification v1.1.0*, 2010.
- [50] WOOLLEY, R., MURRAY, M., DOUNIN, M., AND ERMILOV, R. arc4random(9): predictable sequence vulnerability.

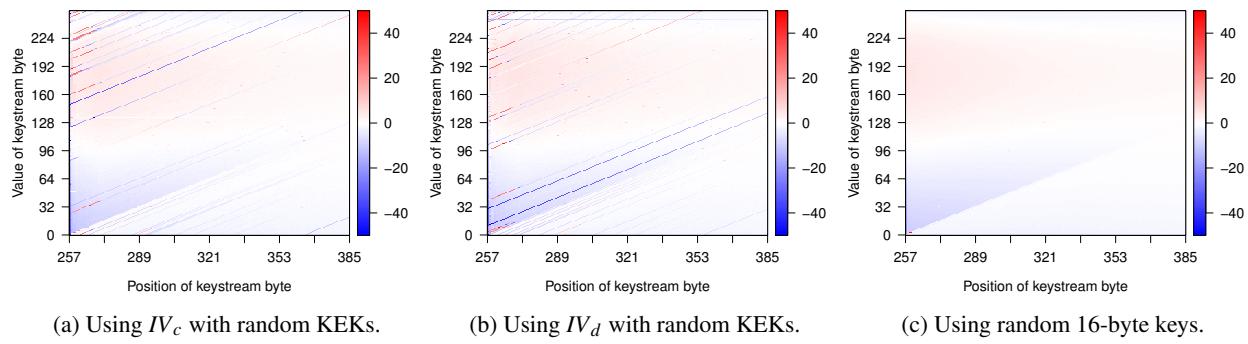


Figure 9: Biases in the RC4 keystream when concatenating a fixed 16-byte IV with a random 16-byte key (here called KEK key), and when using random 16-byte keys. Each points encodes a bias as the number $(pr - 2^{-8}) \cdot 2^{24}$, capped to values in $[-50, 50]$, with pr the empirical probability of the keystream byte value (y-axis) at a given location (x-axis).

DROWN: Breaking TLS using SSLv2

Nimrod Aviram¹, Sebastian Schinzel², Juraj Somorovsky³, Nadia Heninger⁴, Maik Dankel²,
Jens Steube⁵, Luke Valenta⁴, David Adrian⁶, J. Alex Halderman⁶, Viktor Dukhovni⁷,
Emilia Käspér⁸, Shaanan Cohney⁴, Susanne Engels³, Christof Paar³ and Yuval Shavitt¹

¹Department of Electrical Engineering, Tel Aviv University

²Münster University of Applied Sciences

³Horst Görtz Institute for IT Security, Ruhr University Bochum

⁴University of Pennsylvania

⁵Hashcat Project

⁶University of Michigan

⁷Two Sigma/OpenSSL

⁸Google/OpenSSL

Abstract

We present DROWN, a novel cross-protocol attack on TLS that uses a server supporting SSLv2 as an oracle to decrypt modern TLS connections.

We introduce two versions of the attack. The more general form exploits multiple unnoticed protocol flaws in SSLv2 to develop a new and stronger variant of the Bleichenbacher RSA padding-oracle attack. To decrypt a 2048-bit RSA TLS ciphertext, an attacker must observe 1,000 TLS handshakes, initiate 40,000 SSLv2 connections, and perform 2^{50} offline work. The victim client never initiates SSLv2 connections. We implemented the attack and can decrypt a TLS 1.2 handshake using 2048-bit RSA in under 8 hours, at a cost of \$440 on Amazon EC2. Using Internet-wide scans, we find that 33% of all HTTPS servers and 22% of those with browser-trusted certificates are vulnerable to this protocol-level attack due to widespread key and certificate reuse.

For an even cheaper attack, we apply our new techniques together with a newly discovered vulnerability in OpenSSL that was present in releases from 1998 to early 2015. Given an unpatched SSLv2 server to use as an oracle, we can decrypt a TLS ciphertext in one minute on a single CPU—fast enough to enable man-in-the-middle attacks against modern browsers. We find that 26% of HTTPS servers are vulnerable to this attack.

We further observe that the QUIC protocol is vulnerable to a variant of our attack that allows an attacker to impersonate a server indefinitely after performing as few as 2^{17} SSLv2 connections and 2^{58} offline work.

We conclude that SSLv2 is not only weak, but actively harmful to the TLS ecosystem.

1 Introduction

TLS [13] is one of the main protocols responsible for transport security on the modern Internet. TLS and its precursor SSLv3 have been the target of a large number of cryptographic attacks in the research community, both on popular implementations and the protocol itself [33]. Prominent recent examples include attacks on outdated or deliberately weakened encryption in RC4 [3], RSA [5], and Diffie-Hellman [1], different side channels including Lucky13 [2], BEAST [14], and POODLE [35], and several attacks on invalid TLS protocol flows [5, 6, 12].

Comparatively little attention has been paid to the SSLv2 protocol, likely because the known attacks are so devastating and the protocol has long been considered obsolete. Wagner and Schneier wrote in 1996 that their attacks on SSLv2 “will be irrelevant in the long term when servers stop accepting SSL 2.0 connections” [41]. Most modern TLS clients do not support SSLv2 at all. Yet in 2016, our Internet-wide scans find that out of 36 million HTTPS servers, 6 million (17%) support SSLv2.

A Bleichenbacher attack on SSLv2. Bleichenbacher’s padding oracle attack [8] is an adaptive chosen ciphertext attack against PKCS#1 v1.5, the RSA padding standard used in SSL and TLS. It enables decryption of RSA ciphertexts if a server distinguishes between correctly and incorrectly padded RSA plaintexts, and was termed the “million-message attack” upon its introduction in 1998, after the number of decryption queries needed to deduce a plaintext. All widely used SSL/TLS servers include countermeasures against Bleichenbacher attacks.

Our first result shows that the SSLv2 protocol is fatally vulnerable to a form of Bleichenbacher attack that enables

decryption of RSA ciphertexts. We develop a novel application of the attack that allows us to use a server that supports SSLv2 as an efficient padding oracle. This attack is a protocol-level flaw in SSLv2 that results in a feasible attack for 40-bit export cipher strengths, and in fact abuses the universally implemented countermeasures against Bleichenbacher attacks to obtain a decryption oracle.

We also discovered multiple implementation flaws in commonly deployed OpenSSL versions that allow an extremely efficient instantiation of this attack.

Using SSLv2 to break TLS. Second, we present a novel *cross-protocol attack* that allows an attacker to break a passively collected RSA key exchange for any TLS server if the RSA keys are also used for SSLv2, possibly on a different server. We call this attack DROWN (*Decrypting RSA using Obsolete and Weakened eNcryption*).

In its *general* version, the attack exploits the protocol flaws in SSLv2, does not rely on any particular library implementation, and is feasible to carry out in practice by taking advantage of commonly supported export-grade ciphers. In order to decrypt one TLS session, the attacker must passively capture about 1,000 TLS sessions using RSA key exchange, make 40,000 SSLv2 connections to the victim server, and perform 2^{50} symmetric encryption operations. We successfully carried out this attack using an optimized GPU implementation and were able to decrypt a 2048-bit RSA ciphertext in less than 18 hours on a GPU cluster and less than 8 hours using Amazon EC2.

We found that 11.5 million HTTPS servers (33%) are vulnerable to this attack, because many HTTPS servers that do not directly support SSLv2 share RSA keys with other services that do. Of servers offering HTTPS with browser-trusted certificates, 22% are vulnerable.

We also present a *special* version of DROWN that exploits flaws in OpenSSL for a more efficient oracle. It requires roughly the same number of captured TLS sessions as the general attack, but only half as many connections to the victim server and no large computations. This attack can be completed on a single core on commodity hardware in less than a minute, and is limited primarily by how fast the server can complete handshakes. It is fast enough that an attacker can perform man-in-the-middle attacks on live TLS sessions before the handshake times out, and downgrade a modern TLS client to RSA key exchange with a server that prefers non-RSA cipher suites. Our Internet-wide scans suggest that 79% of HTTPS servers that are vulnerable to the general attack, or 26% of all HTTPS servers, are also vulnerable to real-time attacks exploiting these implementation flaws.

Our results highlight the risk that continued support for SSLv2 imposes on the security of much more recent TLS versions. This is an instance of a more general phenomenon of insufficient domain separation, where older, vulnerable security standards can open the door to

attacks on newer versions. We conclude that phasing out outdated and insecure standards should become a priority for standards designers and practitioners.

Disclosure. DROWN was assigned CVE-2016-0800. We disclosed our attacks to OpenSSL and worked with them to coordinate further disclosures. The specific OpenSSL vulnerabilities we discovered have been designated CVE-2015-3197, CVE-2016-0703, and CVE-2016-0704. In response to our findings, OpenSSL has made it impossible to configure a TLS server in such a way that it is vulnerable to DROWN. Microsoft had already disabled SSLv2 for all supported versions of IIS. We also disclosed the attack to the NSS developers, who have disabled SSLv2 on the last NSS tool that supported it and have hastened efforts to entirely remove the protocol from their codebase. In response to our disclosure, Google will disable QUIC support for non-whitelisted servers and modify the QUIC standard. We also notified IBM, Cisco, Amazon, the German CERT-Bund, and the Israeli CERT.

Online resources. Contact information, server test tools, and updates are available at <https://drownattack.com>.

2 Background

In the following, $a||b$ denotes concatenation of strings a and b . $a[i]$ references the i -th byte in a . (N, e) denotes an RSA public key, where N has byte-length ℓ_m ($|N| = \ell_m$) and e is the public exponent. The corresponding secret exponent is $d = 1/e \bmod \phi(N)$.

2.1 PKCS#1 v1.5 encryption padding

Our attacks rely on the structure of RSA PKCS#1 v1.5 padding. Although RSA PKCS#1 v2.0 implements OAEP, SSL/TLS still uses PKCS#1 v1.5. The PKCS#1 v1.5 encryption padding scheme [27] randomizes encryptions by prepending a random padding string PS to a message k (here, a symmetric session key) before RSA encryption:

1. The plaintext message is k , $\ell_k = |k|$. The encrypter generates a random byte string PS , where $|PS| \geq 8$, $|PS| = \ell_m - 3 - \ell_k$, and $0x00 \notin \{PS[1], \dots, PS[|PS|]\}$.
2. The encryption block is $m = 00||02||PS||00||k$.
3. The ciphertext is computed as $c = m^e \bmod N$.

To decrypt such a ciphertext, the decrypter first computes $m = c^d \bmod N$. Then it checks whether the decrypted message m is correctly formatted as a PKCS#1 v1.5-encoded message. We say that the ciphertext c and the decrypted message bytes $m[1]||m[2]||\dots||m[\ell_m]$ are PKCS#1 v1.5 conformant if:

$$\begin{aligned} m[1]||m[2] &= 0x00||0x02 \\ 0x00 &\notin \{m[3], \dots, m[10]\} \end{aligned}$$

If this condition holds, the decrypter searches for the first

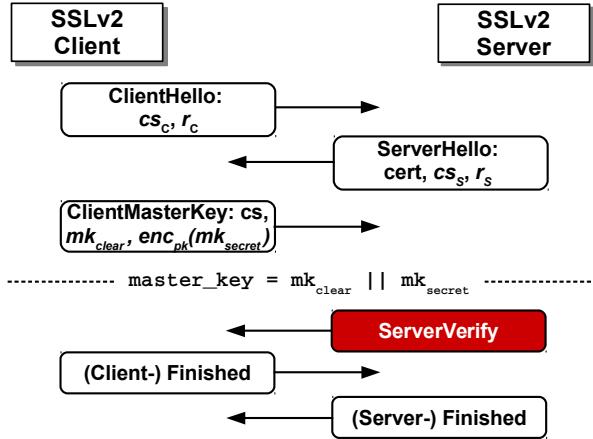


Figure 1: **SSLv2 handshake.** The server responds with a `ServerVerify` message directly after receiving an RSA-PKCS#1 v1.5 ciphertext contained in `ClientMasterKey`. This protocol feature enables our attack.

value $i > 10$ such that $m[i] = 0x00$. Then, it extracts $k = m[i+1] \parallel \dots \parallel m[\ell_m]$. Otherwise, the ciphertext is rejected.

In SSLv3 and TLS, RSA PKCS#1 v1.5 is used to encapsulate the premaster secret exchanged during the hand-shake [13]. Thus, k is interpreted as the premaster secret. In SSLv2, RSA PKCS#1 v1.5 is used for encapsulation of an equivalent key denoted the `master_key`.

2.2 SSL and TLS

The first incarnation of the TLS protocol was the SSL (Secure Socket Layer) protocol, which was designed by Netscape in the 90s. The first two versions of SSL were immediately found to be vulnerable to trivial attacks [40, 41] which were fixed in SSLv3 [17]. Later versions of the standard were renamed TLS, and share a similar structure to SSLv3. The current version of the protocol is TLS 1.2; TLS 1.3 is currently under development.

An SSL/TLS protocol flow consists of two phases: handshake and application data exchange. In the first phase, the communicating parties agree on cryptographic algorithms and establish shared keys. In the second phase, these keys are used to protect the confidentiality and authenticity of the transmitted application data.

The handshake protocol was fundamentally redesigned in the SSLv3 version. This new handshake protocol was then used in later TLS versions up to TLS 1.2. In the following, we describe the RSA-based handshake protocols used in TLS and SSLv2, and highlight their differences.

The SSLv2 handshake protocol. The SSLv2 protocol description [22] is less formally specified than modern RFCs. Figure 1 depicts an SSLv2 handshake. A client initiates an SSLv2 handshake by sending a `ClientHello` message, which includes a list of cipher suites cs_c

supported by the client and a client nonce r_c , termed challenge. The server responds with a `ServerHello` message, which contains a list of cipher suites cs_s supported by the server, the server certificate, and a server nonce r_s , termed `connection_ID`.

The client responds with a `ClientMasterKey` message, which specifies a cipher suite supported by both peers and key data used for constructing a `master_key`. In order to support *export* cipher suites with 40-bit security (e.g., `SSL_RC2_128_CBC_EXPORT40_WITH_MD5`), the key data is divided into two parts:

- mk_{clear} : A portion of the `master_key` sent in the `ClientMasterKey` message as plaintext (termed `clear_key_data` in the SSLv2 standard).
 - mk_{secret} : A secret portion of the `master_key`, encrypted with RSA PKCS#1 v1.5 (termed `secret_key_data`).

The resulting `master_key` mk is constructed by concatenating these two keys: $mk = mk_{clear} || mk_{secret}$. For 40-bit export cipher suites, mk_{secret} is five bytes in length. For non-export cipher suites, the whole `master_key` is encrypted, and the length of mk_{clear} is zero.

The client and server can then compute session keys from the reconstructed `master_key` mk :

$$\begin{aligned} \text{server_write_key} &= MD5(mk||"0"||r_c||r_s) \\ \text{client_write_key} &= MD5(mk||"1"||r_c||r_s) \end{aligned}$$

The server responds with a `ServerVerify` message consisting of the challenge r_c encrypted with the `server_write_key`. Both peers then exchange `Finished` messages in order to authenticate to each other.

Our attack exploits the fact that the server always decrypts an RSA-PKCS#1 v1.5 ciphertext, computes the `server_write_key`, and *immediately* responds with a `ServerVerify` message. The SSLv2 standard implies this message ordering, but does not make it explicit. However, we observed this behavior in every implementation we examined. Our attack also takes advantage of the fact that the encrypted mk_{secret} portion of the `master_key` can vary in length, and is only five bytes for export ciphers.

The TLS handshake protocol. In TLS [13] or SSLv3, the client initiates the handshake with a `ClientHello`, which contains a client random r_c and a list of supported cipher suites. The server chooses one of the cipher suites and responds with three messages, `ServerHello`, `Certificate`, and `ServerHelloDone`. These messages include the server's choice of cipher suite, server nonce r_s , and a server certificate with an RSA public key. The client then uses the public key to encrypt a newly generated 48-byte premaster secret pms and sends it to the server in a `ClientKeyExchange` message. The client and server then derive encryption and MAC keys from the premaster secret and the client and server random nonces. The details of this derivation are not important to our attack. The

client then sends `ChangeCipherSpec` and `Finished` messages. The `Finished` message authenticates all previous handshake messages using the derived keys. The server responds with its own `ChangeCipherSpec` and `Finished` messages.

The two main details relevant to our attacks are:

- The premaster secret is always 48 bytes long, independent of the chosen cipher suite. This is also true for export cipher suites.
- After receiving the `ClientKeyExchange` message, the server waits for the `ClientFinished` message, in order to authenticate the client.

2.3 Bleichenbacher’s attack

Bleichenbacher’s attack is a padding oracle attack—it exploits the fact that RSA ciphertexts should decrypt to PKCS#1 v1.5-compliant plaintexts. If an implementation receives an RSA ciphertext that decrypts to an invalid PKCS#1 v1.5 plaintext, it might naturally leak this information via an error message, by closing the connection, or by taking longer to process the error condition. This behavior can leak information about the plaintext that can be modeled as a cryptographic *oracle* for the decryption process. Bleichenbacher [8] demonstrated how such an oracle could be exploited to decrypt RSA ciphertexts.

Algorithm. In the simplest attack scenario, the attacker has a valid PKCS#1 v1.5 ciphertext c_0 that they wish to decrypt to discover the message m_0 . They have no access to the private RSA key, but instead have access to an oracle \mathcal{O} that will decrypt a ciphertext c and inform the attacker whether the most significant two bytes match the required value for a correct PKCS#1 v1.5 padding:

$$\mathcal{O}(c) = \begin{cases} 1 & \text{if } m = c^d \bmod N \text{ starts with 0x00 02} \\ 0 & \text{otherwise.} \end{cases}$$

If the oracle answers with 1, the attacker knows that $2B \leq m \leq 3B - 1$, where $B = 2^{8(\ell_m - 2)}$. The attacker can take advantage of RSA malleability to generate new candidate ciphertexts for any s :

$$c = (c_0 \cdot s^e) \bmod N = (m_0 \cdot s)^e \bmod N$$

The attacker queries the oracle with c . If the oracle responds with 0, the attacker increments s and repeats the previous step. Otherwise, the attacker learns that for some r , $2B \leq m_0 s - rN < 3B$. This allows the attacker to reduce the range of possible solutions to:

$$\frac{2B + rN}{s} \leq m_0 < \frac{3B + rN}{s}$$

The attacker proceeds by refining guesses for s and r values and successively decreasing the size of the interval containing m_0 . At some point the interval will contain a single valid value, m_0 . Bleichenbacher’s original paper describes this process in further detail [8].

Countermeasures. In order to protect against this attack, the decrypter must not leak information about the PKCS#1 v1.5 validity of the ciphertext. The ciphertext does not decrypt to a valid message, so the decrypter generates a fake plaintext and continues the protocol with this decoy. The attacker should not be able to distinguish the resulting computation from a correctly decrypted ciphertext.

In the case of SSL/TLS, the server generates a random premaster secret to continue the handshake if the decrypted ciphertext is invalid. The client will not possess the session key to send a valid `ClientFinished` message and the connection will terminate.

3 Breaking TLS with SSLv2

In this section, we describe our cross-protocol DROWN attack that uses an SSLv2 server as an oracle to efficiently decrypt TLS connections. The attacker learns the session key for targeted TLS connections but does not learn the server’s private RSA key. We first describe our techniques using a generic SSLv2 oracle. In Section 4.1, we show how a protocol flaw in SSLv2 can be used to construct such an oracle, and describe our general DROWN attack. In Section 5, we show how an implementation flaw in common versions of OpenSSL leads to a more powerful oracle and describe our efficient special DROWN attack.

We consider a server accepting TLS connections from clients. The connections are established using a secure, state-of-the-art TLS version (1.0–1.2) and a `TLS_RSA` cipher suite with a private key unknown to the attacker.

The same RSA public key as the TLS connections is also used for SSLv2. For simplicity, our presentation will refer to the servers accepting TLS and SSLv2 connections as the same entity.

Our attacker is able to passively eavesdrop on traffic between the client and server and record RSA-based TLS traffic. The attacker may or may not be also required to perform active man-in-the-middle interference, as explained below.

The attacker can expect to decrypt one out of 1,000 intercepted TLS connections in our attack for typical parameters. This is a devastating threat in many scenarios. For example, a decrypted TLS connection might reveal a client’s HTTP cookie or plaintext password, and an attacker would only need to successfully decrypt a single ciphertext to compromise the client’s account. In order to collect 1,000 TLS connections, the attacker might simply wait patiently until sufficiently many connections are recorded. A less patient attacker might use man-in-the-middle interference, as in the BEAST attack [14].

3.1 A generic SSLv2 oracle

Our attacks make use of an oracle that can be queried on a ciphertext and leaks information about the decrypted plaintext; this abstractly models the information gained

from an SSLv2 server’s behavior. Our SSLv2 oracles reveal many bytes of plaintext, enabling an efficient attack.

Our cryptographic oracle \mathcal{O} has the following functionality: \mathcal{O} decrypts an RSA ciphertext c and responds with ciphertext validity based on the decrypted message m . The ciphertext is valid only if m starts with 0x00 02 followed by non-null padding bytes, a delimiter byte 0x00, and a master key mk_{secret} of correct byte length ℓ_k . We call such a ciphertext *SSLv2 conformant*.

All of the SSLv2 padding oracles we instantiate give the attacker similar information about a PKCS#1 v1.5 conformant SSLv2 ciphertext:

$$\mathcal{O}(c) = \begin{cases} mk_{secret} & \text{if } c^d \bmod N = 00||02||PS||00||mk_{secret} \\ 0 & \text{otherwise.} \end{cases}$$

That is, the oracle $\mathcal{O}(c)$ will return the decrypted message mk_{secret} if it is queried on a PKCS#1 v1.5 conformant SSLv2 ciphertext c corresponding to a correctly PKCS#1 v1.5 padded encryption of mk_{secret} . The attacker then learns $\ell_k + 3$ bytes of $m = c^d \bmod N$: the first two bytes are 00||02, and the last $\ell_k + 1$ bytes are 00|| mk_{secret} . The length ℓ_k of mk_{secret} varies based on the cipher suite used to instantiate the oracle. For export-grade cipher suites such as `SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5`, k will be 5 bytes, so the attacker learns 8 bytes of m .

3.2 DROWN attack template

Our attacker will use an SSLv2 oracle \mathcal{O} to decrypt a TLS ClientKeyExchange. The behavior of \mathcal{O} poses two problems for the attacker. First, a TLS key exchange ciphertext decrypts to a 48-byte premaster secret. But since no SSLv2 cipher suites have 48-byte key strengths, this means that a valid TLS ciphertext is invalid to our oracle \mathcal{O} . In order to apply Bleichenbacher’s attack, the attacker must transform the TLS ciphertext into a valid SSLv2 key exchange message. Second, \mathcal{O} is very restrictive, since it strictly checks the length of the unpadded message. According to Bardou et al. [4], Bleichenbacher’s attack would require 12 million queries to such an oracle.¹

Our attacker overcomes these problems by following this generic attack flow:

0. The attacker collects many encrypted TLS RSA key exchange messages.
1. The attacker converts one of the intercepted TLS ciphertexts containing a 48-byte premaster secret to an RSA PKCS#1 v1.5 encoded ciphertext valid to the SSLv2 oracle \mathcal{O} .
2. Once the attacker has obtained a valid SSLv2 RSA ciphertext, they can continue with a modified version of Bleichenbacher’s attack, and decrypt the message after many more oracle queries.

¹See Table 1 in [4]. The oracle is denoted with the term FFF.

3. The attacker then transforms the decrypted plaintext back into the original plaintext, which is one of the collected TLS handshakes.

We describe the algorithmic improvements we use to make each of these steps efficient below.

3.2.1 Finding an SSLv2 conformant ciphertext

The first step for the attacker is to transform the original TLS ClientKeyExchange message c_0 from a TLS conformant ciphertext into an SSLv2 conformant ciphertext.

For this task, we rely on the concept of *trimmers*, which were introduced by Bardou et al. [4]. Assume that the message $m_0 = c_0^d \bmod N$ is divisible by a small number t . In that case, $m_0 \cdot t^{-1} \bmod N$ simply equals the natural number m_0/t . If we choose $u \approx t$, and multiply the original message by $u \cdot t^{-1}$, the resulting number will lie near the original message: $m_0 \approx m_0/t \cdot u$.

This method gives a good chance of generating a new SSLv2 conformant message. Let c_0 be an intercepted TLS conformant RSA ciphertext, and let $m_0 = c_0^d \bmod N$ be the plaintext. We select a multiplier $s = u/t \bmod N = ut^{-1} \bmod N$ where u and t are coprime, compute the value $c_1 = c_0 s^e \bmod N$, and query $\mathcal{O}(c_1)$. We will receive a response if $m_1 = m_0 \cdot u/t$ is SSLv2 conformant.

As an example, let us assume a 2048-bit RSA ciphertext with $\ell_k = 5$, and consider the fraction $u = 7, t = 8$. The probability that $c_0 \cdot u/t$ will be SSLv2 conformant is 1/7,774, so we expect to make 7,774 oracle queries before obtaining a positive response from \mathcal{O} . Appendix A.1 gives more details on computing these probabilities.

3.2.2 Shifting known plaintext bytes

Once we have obtained an SSLv2 conformant ciphertext c_1 , the oracle has also revealed the $\ell_k + 1$ least significant bytes (mk_{secret} together with the delimiter byte 0x00) and two most significant 0x00 02 bytes of the SSLv2 conformant message m_1 . We would like to *rotate* these known bytes around to the right, so that we have a large block of contiguous known most significant bytes of plaintext. In this section, we show that this can be accomplished by multiplying by some shift $2^{-r} \bmod N$. In other words, given an SSLv2 conformant ciphertext $c_1 = m_1^e \bmod N$, we can efficiently generate an SSLv2 conformant ciphertext $c_2 = m_2^e \bmod N$ where $m_2 = s \cdot m_1 \cdot 2^{-r} \bmod N$ and we know several most significant bytes of m_2 .

Let $R = 2^{8(k+1)}$ and $B = 2^{8(\ell_m-2)}$. Abusing notation slightly, let the integer $m_1 = 2 \cdot B + PS \cdot R + mk_{secret}$ be the plaintext satisfying $m_1^e = c_1 \bmod N$. At this stage, the ℓ_k -byte integer mk_{secret} is known and the $\ell_m - \ell_k - 3$ -byte integer PS is not.

Let $\tilde{m}_1 = 2 \cdot B + mk_{secret}$ be the known components of m_1 , so $m_1 = \tilde{m}_1 + PS \cdot R$. We can use this to compute a new plaintext for which we know many most significant

bytes. Consider the value:

$$m_1 \cdot R^{-1} \bmod N = \tilde{m}_1 \cdot R^{-1} + PS \bmod N.$$

The value of PS is unknown and consists of $\ell_m - \ell_k - 3$ bytes. This means that the known value $\tilde{m}_1 \cdot R^{-1}$ shares most of its $\ell_k + 3$ most significant bytes with $m_1 \cdot R^{-1}$.

Furthermore, we can iterate this process by finding a new multiplier s such that $m_2 = s \cdot m_1 \cdot R^{-1} \bmod N$ is also SSLv2 conformant. A randomly chosen $s < 2^{30}$ will work with probability $2^{-25.4}$. We can take use the bytes we have already learned about m_1 to efficiently compute such an s with only 678 oracle queries in expectation for a 2048-bit RSA modulus. Appendix A.3 gives more details.

3.2.3 Adapted Bleichenbacher iteration

It is feasible for all of our oracles to use the previous technique to entirely recover a plaintext message. However, for our SSLv2 protocol oracle it is cheaper after a few iterations to continue using Bleichenbacher’s original attack. We can apply the original algorithm proposed by Bleichenbacher as described in Section 2.3.

Each step obtains a message that starts with the required 0x00 02 bytes after two queries in expectation. Since we know the value of the $\ell_k + 1$ least significant bytes after multiplying by any integer, we can query the oracle only on multipliers that cause the $(\ell_k + 1)$ st least significant byte to be zero. However, we cannot ensure that the padding string is entirely nonzero; for a 2048-bit modulus this will hold with probability 0.37.

For a 2048-bit modulus, the total expected number of queries when using this technique to fully decrypt the plaintext is $2048 * 2 / 0.37 \approx 11,000$.

4 General DROWN

In this section, we describe how to use any correct SSLv2 implementation accepting export-grade cipher suites as a padding oracle. We then show how to adapt the techniques described in Section 3.2 to decrypt TLS RSA ciphertexts.

4.1 The SSLv2 export padding oracle

SSLv2 is vulnerable to a direct message side channel vulnerability exposing a Bleichenbacher oracle to the attacker. The vulnerability follows from three properties of SSLv2. First, the server immediately responds with a `ServerVerify` message after receiving the `ClientMasterKey` message, which includes the RSA ciphertext, without waiting for the `ClientFinished` message that proves the client knows the RSA plaintext. Second, when choosing 40-bit export RC2 or RC4 as the symmetric cipher, only 5 bytes of the `master_key` (mk_{secret}) are sent encrypted using RSA, and the remaining 11 bytes are sent in cleartext. Third, a server implementation that correctly implements the anti-Bleichenbacher countermeasure and receives an RSA key exchange message with invalid padding will generate a random premaster secret

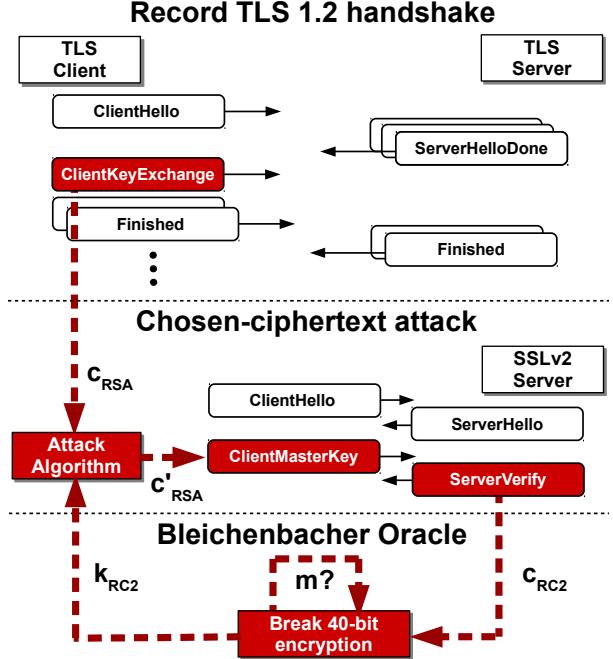


Figure 2: **SSLv2-based Bleichenbacher attack on TLS.** An attacker passively collects RSA ciphertexts from a TLS 1.2 handshake, and then performs oracle queries against a server that supports SSLv2 with the same public key to decrypt the TLS ciphertext.

and carry out the rest of the TLS handshake using this randomly generated key material.

This allows an attacker to deduce the validity of RSA ciphertexts in the following manner:

1. The attacker sends a `ClientMasterKey` message, which contains an RSA ciphertext c_0 and any choice of 11 clear key bytes for mk_{clear} . The server responds with a `ServerVerify` message, which contains the challenge encrypted using the `server_write_key`.
2. The attacker performs an *exhaustive search* over the possible values of the 5 bytes of the `master_key` mk_{secret} , computes the corresponding `server_write_key`, and checks whether the `ServerVerify` message decrypts to `challenge`. One value should pass this check; call it mk_0 . Recall that if the RSA plaintext was valid, mk_0 is the unpadded data in the RSA plaintext c_0^d . Otherwise, mk_0 is a randomly generated sequence of 5 bytes.
3. The attacker re-connects to the server with the same RSA ciphertext c_0 . The server responds with another `ServerVerify` message that contains the current challenge encrypted using the current `server_write_key`. If the decrypted RSA cipher-

text was valid, the attacker can use mk_0 to decrypt a correct challenge value from the ServerVerify message. Otherwise, if the ServerVerify message does not decrypt to challenge, the RSA ciphertext was invalid, and mk_0 must have been random.

Thus we can instantiate an oracle $\mathcal{O}_{\text{SSLv2-export}}$ using the procedure above; each oracle query requires two server connections and 2^{40} decryption attempts in the simplest case. For each oracle call $\mathcal{O}_{\text{SSLv2-export}}(c)$, the attacker learns whether c is valid, and if so, learns the two most significant bytes 0x00 02, the sixth least significant 0x00 delimiter byte, and the value of the 5 least significant bytes of the plaintext m .

4.2 TLS decryption attack

In this section, we describe how the oracle described in Section 4.1 can be used to carry out a feasible attack to decrypt passively collected TLS ciphertexts.

As described in Section 3, we consider a server that accepts TLS connections from clients using an RSA public key that is exposed via SSLv2, and an attacker who is able to passively observe these connections.

We also assume the server supports export cipher suites for SSLv2. This can happen for two reasons. First, the same server operators that fail to follow best practices in disabling SSLv2 [40] may also fail to follow best practices by supporting export cipher suites. Alternatively, the server might be running a version of OpenSSL prior to January 2016, in which case it is vulnerable to the OpenSSL cipher suite selection bug described in Section 7, and an attacker may negotiate a cipher suite of his choice independent of the server configuration.

The attacker needs access to computing power sufficient to perform a 2^{50} time attack, mostly brute forcing symmetric key encryption. After our optimizations, this can be done with a one-time investment of a few thousand dollars of GPUs, or in a few hours for a few hundred dollars in the cloud. Our cost estimates are described in Section 4.3.

4.2.1 Constructing the attack

The attacker can exploit the SSLv2 vulnerability following the generic attack outline described in Section 3.2, consisting of several distinct phases:

0. The attacker passively collects 1,000 TLS handshakes from connections using RSA key exchange.
1. They then attempt to convert the intercepted TLS ciphertexts containing a 48-byte premaster secret to valid RSA PKCS#1 v1.5 encoded ciphertexts containing five-byte messages using the fractional trimmers described in Section 3.2.1, and querying $\mathcal{O}_{\text{SSLv2-export}}$. The attacker sends the modified ciphertexts to the server using fresh SSLv2 connections with weak symmetric ciphers and uses the

ServerVerify messages to deduce ciphertext validity as described in the previous section. For each queried RSA ciphertext, the attacker must perform a brute force attack on the weak symmetric cipher. The attacker expects to obtain a valid SSLv2 ciphertext after roughly 10,000 oracle queries, or 20,000 connections to the server.

2. Once the attacker has obtained a valid SSLv2 RSA ciphertext $c_1 = m_1^e$, they use the shifting technique explained in Section 3.2.2 to find an integer s_1 such that $m_2 = m_1 \cdot 2^{-40} \cdot s_1$ is also SSLv2 conformant. Appendix A.4 contains more details on this step.
3. The attacker then applies the shifting technique again to find another integer s_2 such that $m_3 = m_2 \cdot 2^{-40} \cdot s_2$ is also SSLv2 conformant.
4. They then search for yet another integer s_3 such that $m_3 \cdot s_3$ is also SSLv2 conformant.
5. Finally, the attacker can continue with our adapted Bleichenbacher iteration technique described in Section 3.2.3, and decrypts the message after an expected 10,000 additional oracle queries, or 20,000 connections to the server.
6. The attacker can then transform the decrypted plaintext back into the original plaintext, which is one of the 1,000 intercepted TLS handshakes.

The rationale behind the different phases. Bleichenbacher’s original algorithm requires a conformant message m_0 , and a multiplier s_1 such that $m_1 = m_0 \cdot s_1$ is also conformant. Naïvely, it would appear we can apply the same algorithm here, after completing Phase 1. However, the original algorithm expects s_1 to be of size about 2^{24} . This is not the case when we use fractions for s_1 , as the integer $s_1 = ut^{-1} \bmod N$ will be the same size as N .

Therefore, our approach is to find a conformant message for which we know the 5 most significant bytes; this will happen after multiple rotations and this message will be m_3 . After finding such a message, finding s_3 such that $m_4 = m_3 \cdot s_3$ is also conformant becomes trivial. From there, we can finally apply the adapted Bleichenbacher iteration technique as described in Appendix A.5.

4.2.2 Attack performance

The attacker wishes to minimize three major costs in the attack: the number of recorded ciphertexts from the victim client, the number of connections to the victim server, and the number of symmetric keys to be brute forced. The requirements for each of these elements are governed by the set of fractions to be multiplied with each RSA ciphertext in the first phase, as described in Section 3.2.1.

Table 1 highlights a few choices for F and the resulting performance metrics for 2048-bit RSA keys. Appendix A provides more details on the derivation of these numbers

Optimizing for	Cipher-texts	$ F $	SSLv2 connections	Offline work
offline work	12,743	1	50,421	$2^{49.64}$
offline work	1,055	10	46,042	$2^{50.63}$
compromise	4,036	2	41,081	$2^{49.98}$
online work	2,321	3	38,866	$2^{51.99}$
online work	906	8	39,437	$2^{52.25}$

Table 1: **2048-bit Bleichenbacher attack complexity.**

The cost to decrypt one ciphertext can be adjusted by choosing the set of fractions F the attacker applies to each of the passively collected ciphertexts in the first step of the attack. This choice affects several parameters: the number of these collected ciphertexts, the number of connections the attacker makes to the SSLv2 server, and the number of offline decryption operations.

Key size	Phase 1	Phases 2–5	Total queries	Offline work
1024	4,129	4,132	8,261	$2^{50.01}$
2048	6,919	12,468	19,387	$2^{50.76}$
4096	18,286	62,185	80,471	$2^{52.16}$

Table 2: **Oracle queries required by our attack.** In Phase 1, the attacker queries the oracle until an SSLv2 conformant ciphertext is found. In Phases 2–5, the attacker decrypts this ciphertext using leaked plaintext. These numbers minimize total queries. In our attack, an oracle query represents two server connections.

and other optimization choices. Table 2 gives the expected number of Bleichenbacher queries for different RSA key sizes, when minimizing total oracle queries.

4.3 Implementing general DROWN with GPUs

The most computationally expensive part of our general DROWN attack is breaking the 40-bit symmetric key, so we developed a highly optimized GPU implementation of this brute force attack. Our first naïve GPU implementation performed around 26MH/s, where MH denotes the time required for testing one million possible values of mk_{secret} . Our optimized implementation runs at a final speed of 515MH/s, a speedup factor of 19.8.

We obtained our improvements through a number of optimizations. For example, our original implementation ran into a communication bottleneck in the PCI-E bus in transmitting candidate keys from CPU to GPU, so we removed this bottleneck by generating key candidates on the GPU itself. We optimized memory management, including storing candidate keys and the RC2 permutation table in constant memory, which is almost as fast as a register, instead of slow global memory.

We experimentally evaluated our optimized implementation on a local cluster and in the cloud. We used it to execute a full attack of $2^{49.6}$ tested keys on each platform. The required number of keys to test during the attack is a random variable, distributed geometrically, with an expectation that ranges between $2^{49.6}$ and $2^{52.5}$ depending on the choice of optimization parameters. We treat a full attack as requiring $2^{49.6}$ tested keys overall.

Hashcat. Hashcat [20] is an open source optimized password-recovery tool. The Hashcat developers allowed us to use their GPU servers for our attack evaluation. The servers contain a total of 40 GPUs: 32 Nvidia GTX 980 cards, and 8 AMD R9 290X cards. The value of this equipment is roughly \$18,040. Our full attack took less than 18 hours to complete on the Hashcat servers, with the longest single instance taking 17h9m.

Amazon EC2. We also ran our optimized GPU code on the Amazon Elastic Compute Cloud (EC2) service. We used a cluster composed of 200 variable-price “spot” instances: 150 g2.2xlarge instances, each containing one high-performance NVIDIA GPU with 1,536 CUDA cores and 50 g2.8xlarge instances, each containing four of these GPUs. When we ran our experiments in January 2016, the average spot rates we paid were \$0.09/hr and \$0.83/hr respectively. Our full attack finished in under 8 hours including startup and shutdown for a cost of \$440.

4.4 OpenSSL SSLv2 cipher suite selection bug

General DROWN is a protocol flaw, but the population of vulnerable hosts is increased due to a bug in OpenSSL that causes many servers to erroneously support SSLv2 and export ciphers even when configured not to. The OpenSSL team intended to disable SSLv2 by default in 2010, with a change that removed all SSLv2 cipher suites from the default list of ciphers offered by the server [36]. However, the code for the protocol itself was not removed in standard builds and SSLv2 itself remained enabled. We discovered a bug in OpenSSL’s SSLv2 cipher suite negotiation logic that allows clients to select SSLv2 cipher suites even when they are not explicitly offered by the server. We notified the OpenSSL team of this vulnerability, which was assigned CVE-2015-3197. The problem was fixed in OpenSSL releases 1.0.2f and 1.0.1r [36].

5 Special DROWN

We discovered multiple vulnerabilities in recent (but not current) versions of the OpenSSL SSLv2 handshake code that create even more powerful Bleichenbacher oracles, and drastically reduce the amount of computation required to implement our attacks. The vulnerabilities, designated CVE-2016-0703 and CVE-2016-0704, were present in the OpenSSL codebase from at least the start of the repository, in 1998, until they were unknowingly fixed on March

4, 2015 by a patch [28] designed to correct an unrelated problem [11]. By adapting DROWN to exploit this special case, we can significantly cut both the number of connections and the computational work required.

5.1 The OpenSSL “extra clear” oracle

Prior to the fix, OpenSSL servers improperly allowed the `ClientMasterKey` message to contain `clear_key_data` bytes for *non-export* ciphers. When such bytes are present, the server substitutes them for bytes from the encrypted key. For example, consider the case that the client chooses a 128-bit cipher and sends a 16-byte encrypted key $k[1], k[2], \dots, k[16]$ but, contrary to the protocol specification, includes 4 null bytes of `clear_key_data`. Vulnerable OpenSSL versions will construct the following `master_key`:

```
[00 00 00 00 k[1] k[2] k[3] k[4] ... k[9] k[10] k[11] k[12]]
```

This enables a straightforward key recovery attack against such versions. An attacker that has intercepted an SSLv2 connection takes the RSA ciphertext of the encrypted key and replays it in non-export handshakes to the server with varying lengths of `clear_key_data`. For a 16-byte encrypted key, the attacker starts with 15 bytes of clear key, causing the server to use the `master_key`:

```
[00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 k[1]]
```

The attacker can brute force the first byte of the encrypted key by finding the matching `ServerVerify` message among 256 possibilities. Knowing $k[1]$, the attacker makes another connection with the same RSA ciphertext but 14 bytes of clear key, resulting in the `master_key`:

```
[00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 k[1] k[2]]
```

The attacker can now easily brute force $k[2]$. With only 15 probe connections and an expected $15 \cdot 128 = 1,920$ trial encryptions, the attacker learns the entire `master_key` for the recorded session.

As this oracle is obtained by improperly sending unexpected clear-key bytes, we call it the Extra Clear oracle.

This session key-recovery attack can be directly converted to a Bleichenbacher oracle. Given a candidate ciphertext and symmetric key length ℓ_k , the attacker sends the ciphertext with ℓ_k known bytes of `clear_key_data`. The oracle decision is simple:

- If the ciphertext is valid, the `ServerVerify` message will reflect a `master_key` consisting of those ℓ_k known bytes.
- If the ciphertext is invalid, the `master_key` will be replaced with ℓ_k random bytes (by following the countermeasure against the Bleichenbacher attack), resulting in a different `ServerVerify` message.

This oracle decision requires one connection to the server and one `ServerVerify` computation. After the attacker has found a valid ciphertext corresponding to a

ℓ_k -byte encrypted key, they recover the ℓ_k plaintext bytes by repeating the key recovery attack from above. Thus our oracle $\mathcal{O}_{\text{SSLv2-extra-clear}}(c)$ requires one connection to determine whether c is valid. After ℓ_k connections, the attacker additionally learns the ℓ_k least significant bytes of m . We model this as a single oracle call, but the number of server connections will vary depending on the response.

5.2 MITM attack against TLS

Special DROWN is fast enough that it can decrypt a TLS premaster secret *online*, during a connection handshake. A man-in-the-middle attacker can use it to compromise connections between modern browsers and TLS servers—even those configured to prefer non-RSA cipher suites.

The MITM attacker impersonates the server and sends a `ServerHello` message that selects a cipher suite with RSA as the key-exchange method. Then, the attacker uses special DROWN to decrypt the premaster secret. The main difficulty is completing the decryption and producing a valid `ServerFinished` message before the client’s connection times out. Most browsers will allow the handshake to last up to one minute [1].

The attack requires targeting an average of 100 connections, only one of which will be attacked, probabilistically. The simplest way for the attacker to facilitate this is to use JavaScript to cause the client to connect repeatedly to the victim server, as described in Section 3. Each connection is tested against the oracle with only small number of fractions, and the attacker can discern immediately when he receives a positive response from the oracle.

Note that since the decryption must be completed online, the Leaky Export oracle cannot be used, and the attack uses only the Extra Clear oracle.

5.2.1 Constructing the attack

We will use `SSL_DES_192_EDE3_CBC_WITH_MD5` as the cipher suite, allowing the attacker to recover 24 bytes of key at a time. The attack works as follows:

0. The attacker causes the victim client to connect repeatedly to the victim server, with at least 100 connections.
1. The attacker uses the fractional trimmers as described in Section 3.2.1 to convert one of the TLS ciphertexts into an SSLv2 conformant ciphertext c_0 .
2. Once the attacker has obtained a valid SSLv2 ciphertext c_1 , they repeatedly use the shifting technique described in Section 3.2.2 to rotate the message by 25 bytes each iteration, learning 27 bytes with each shift. After several iterations, they have learned the entire plaintext.
3. The attacker then transforms the decrypted SSLv2 plaintext into the decrypted TLS plaintext.

Using 100 fractional trimmers, this more efficient oracle attack allows the attacker to recover one in 100 TLS session keys using only about 27,000 connections to the server, as described in Appendix A.6. The computation cost is so low that we can complete the full attack on a single workstation in under one minute.

5.3 The OpenSSL ‘leaky export’ oracle

In addition to the extra clear implementation bug, the same set of OpenSSL versions also contain a separate bug, where they do not follow the correct algorithm for their implementation of the Bleichenbacher countermeasure. We now describe this faulty implementation:

- The SSLv2 ClientKeyExchange message contains the mk_{clear} bytes immediately before the ciphertext c . Let p be the buffer starting at the first mk_{clear} byte.
- Decrypt c in place. If the decryption operation succeeds, and c decrypted to a plaintext of a correct padded length, p now contains the 11 mk_{clear} bytes followed by the 5 mk_{secret} bytes.
- If c decrypted to an unpadded plaintext k of incorrect length, the decryption operation overwrites the first $j = \min(|k|, 5)$ bytes of c with the first j bytes of k .
- If c is not SSLv2 conformant and the decryption operation failed, randomize the first five bytes of p , which are the first five bytes of mk_{clear} .

This behavior allows the attacker to distinguish between these three cases. Suppose the attacker sends 11 null bytes as mk_{clear} . Then these are the possible cases:

1. c decrypts to a correctly padded plaintext k of the expected length, 5 bytes. Then the following `master_key` will be constructed:

`[00 00 00 00 00 00 00 00 00 00 k[1] k[2] k[3] k[4] k[5]]`

2. c decrypts to a correctly padded plaintext k of a wrong length. Let r be the five random bytes the server generated. The yielded `master_key` will be:

`[r[1] r[2] r[3] r[4] r[5] 00 00 00 00 00 00 k[1] k[2] k[3] k[4] k[5]]`

when $|k| \geq 5$. If $|k| < 5$, the server substitutes the first $|k|$ bytes of c with the first $|k|$ bytes of k . Using $|k| = 3$ as an example, the `master_key` will be:

`[r[1] r[2] r[3] r[4] r[5] 00 00 00 00 00 00 k[1] k[2] k[3] c[4] c[5]]`

3. c is not SSLv2 conformant, and hence the decryption operation failed. The resulting `master_key` will be:

`[r[1] r[2] r[3] r[4] r[5] 00 00 00 00 00 00 c[1] c[2] c[3] c[4] c[5]]`

The attacker detects case (3) by performing an exhaustive search over the 2^{40} possibilities for r , and checking whether any of the resulting values for the `master_key` correctly decrypts the observed `ServerVerify` message. If no r value satisfies this property, then c^d starts with bytes 0x00 02. The attacker then distinguishes between

cases (1) and (2) by performing an exhaustive search over the five bytes of k , and checking whether any of the resulting values for mk correctly decrypts the observed `ServerVerify` message.

As this oracle leaks information when using export ciphers, we have named it the Leaky Export oracle.

In conclusion, $\mathcal{O}_{\text{SSLv2-export-leaky}}$ allows an attacker to obtain a valid oracle response for all ciphertexts which decrypt to a correctly-padded plaintext of *any* length. This is in contrary to the previous oracles $\mathcal{O}_{\text{SSLv2-extra-clear}}$ and $\mathcal{O}_{\text{SSLv2-export}}$, which required the plaintext to be of a specific length. Each oracle query to $\mathcal{O}_{\text{SSLv2-export-leaky}}$ requires one connection to the server and 2^{41} offline work.

Combining the two oracles. The attacker can use the Extra Clear and Leaky Export oracles together in order to reduce the number of queries required for the TLS decryption attack. They first test a TLS conformant ciphertext for divisors using the Leaky Export oracle, then use fractions dividing the plaintext with both oracles. Once the attacker has obtained a valid SSLv2 ciphertext c_1 , they repeatedly use the shifting technique described in Section 3.2.2 to rotate the message by 25 bytes each iteration while choosing 3DES as the symmetric cipher, learning 27 bytes with each shift. After several iterations, they have learned the entire plaintext, using 6,300 queries (again for a 2048-bit modulus). This brings the overall number of queries for this variant of the attack to $900 + 16 * 4 + 6,300 = 7,264$. These parameter choices are not necessarily optimal. We give more details in Appendix A.7.

6 Extending the attack to QUIC

DROWN can also be extended to a feasible-time man-in-the-middle attack against QUIC [26]. QUIC [10, 39] is a recent cryptographic protocol designed and implemented by Google that is intended to reduce the setup time to establish a secure connection while providing security guarantees analogous to TLS. QUIC’s security relies on a static “server config” message signed by the server’s public key. Jager et al. [26] observe that an attacker who can forge a signature on a malicious QUIC server config once would be able to impersonate the server indefinitely. In this section, we show an attacker with significant resources would be able to mount such an attack against a server whose RSA public keys is exposed via SSLv2.

A QUIC client receives a “server config” message, signed by the server’s public key, which enumerates connection parameters: a static elliptic curve Diffie-Hellman public value, and a validity period. In order to mount a man-in-the-middle attack against any client, the attacker wishes to generate a valid server config message containing their own Diffie-Hellman value, and an expiration date far in the future.

The attacker needs to present a forged QUIC config to the client in order to carry out a successful attack. This is

Protocol	Attack type	Oracle	SSLv2 connections	Offline work	See §
TLS	Decrypt	SSLv2	41,081	2^{50}	4.2
TLS	Decrypt	Special	7,264	2^{51}	5.3
TLS	MITM	Special	27,000	2^{15}	5.2
QUIC	MITM	SSLv2	2^{25}	2^{65}	6.1
QUIC	MITM	Special	2^{25}	2^{25}	6.2
QUIC	MITM	Special	2^{17}	2^{58}	6.2

Table 3: **Summary of attacks.** “Oracle” denotes the oracle required to mount each attack, which also implies the vulnerable set of SSLv2 implementations. SSLv2 denotes any SSLv2 implementation, while “Special” denotes an OpenSSL version vulnerable to special DROWN.

straightforward, since QUIC discovery may happen over non-encrypted HTTP [19]. The server does not even need to support QUIC at all: an attacker could impersonate the attacked server over an unencrypted HTTP connection and falsely indicate that the server supports QUIC. The next time the client connects to the server, it will attempt to connect using QUIC, allowing the attacker to present the forged “server config” message and execute the attack [26].

6.1 QUIC signature forgery attack based on general DROWN

The attack proceeds much as in Section 3.2, except that some of the optimizations are no longer applicable, making the attack more expensive.

The first step is to discover a valid, PKCS conformant SSLv2 ciphertext. In the case of TLS decryption, the input ciphertext was PKCS conformant to begin with; this is not the case for the QUIC message c_0 . Thus for the first phase, the attacker iterates through possible multiplier values s until they randomly encounter a valid SSLv2 message in $c_0 \cdot s^d$. For 2048-bit RSA keys, the probability of this random event is $P_{rnd} \approx 2^{-25}$; see Section 3.2.

Once the first SSLv2 conformant message is found, the attacker proceeds with the signature forgery as they would in Step 2 of the TLS decryption attack. The required number of oracle queries for this step is roughly 12,468 for 2048-bit RSA keys.

Attack cost. The overall oracle query cost is dominated by the $2^{25} \approx 34$ million expected queries in the first phase, above. At a rate of 388 queries/second, the attacker would finish in one day; at a rate of 12 queries/second they would finish in one month.

For the SSLv2 export padding oracle, the offline computation to break a 40-bit symmetric key for each query requires iterating over 2^{65} keys. At our optimized GPU implementation rate of 515 million keys per second, this

would require 829,142 GPU days. Our experimental GPU hardware retails for \$400. An investment of \$10 million to purchase 25,000 GPUs would reduce the wall clock time for the attack to 33 days.

Our implementation run on Amazon EC2 processed about 174 billion keys per g2.2xlarge instance-hour, so at a cost of \$0.09/instance-hour the full attack would cost \$9.5 million and could be parallelized to Amazon’s capacity.

6.2 Optimized QUIC signature forgery based on special DROWN

For targeted servers that are vulnerable to special DROWN, we are unaware of a way to combine the two special DROWN oracles; the attacker would have to choose a single oracle which minimizes his subjective cost. For the Extra Clear oracle, there is only negligible computation per oracle query, so the computational cost for the first phase is 2^{25} . For the Leaky Export oracle, as explained below, the required offline work is 2^{58} , and the required number of server connections is roughly 145,573. Both oracles appear to bring this attack well within the means of a moderately provisioned adversary.

Mounting the attack using Leaky Export. For a 2048-bit RSA modulus, the probability of a random message being conformant when querying $\mathcal{O}_{\text{SSLv2-export-leaky}}$ is $P_{rnd} \approx (1/256)^2 * (255/256)^8 * (1 - (255/256)^{246}) \approx 2^{-17}$. Therefore, to compute c^d when c is not SSLv2 conformant, the attacker randomly generates values for s and tests $c \cdot s^e$ against the Leaky Export oracle. After roughly $2^{17} \approx 131,000$ queries, they obtain a positive response, and learn that $c^d \cdot s$ starts with bytes 0x00 02.

Naively, it would seem the attacker can then apply one of the techniques presented in this work, but $\mathcal{O}_{\text{SSLv2-export-leaky}}$ does not provide knowledge of any least significant plaintext bytes when the plaintext length is not at most the correct one. Instead, the attacker proceeds directly according to the algorithm presented in [4]. Referring to Table 1 in [4], $\mathcal{O}_{\text{SSLv2-export-leaky}}$ is denoted with the term FFT, as it returns a positive response for a correctly padded plaintext of any length, and the median number of required queries for this oracle is 14,501. This number of queries is dominated by the 131,000 queries the attacker has already executed. As each query requires testing roughly 2^{41} keys, the required offline work is approximately 2^{58} .

Future changes to QUIC. In addition to disabling QUIC support for non-whitelisted servers, Google have informed us that they plan to change the QUIC standard, so that the “server config” message will include a client nonce to prove freshness. They also plan to limit QUIC discovery to HTTPS.

Protocol	Port	All certificate			Trusted certificates		
		SSL/TLS	SSLv2 support	Vulnerable key	SSL/TLS	SSLv2 support	Vulnerable key
SMTP	25	3,357 K	936 K (28%)	1,666 K (50%)	1,083 K	190 K (18%)	686 K (63%)
POP3	110	4,193 K	404 K (10%)	1,764 K (42%)	1,787 K	230 K (13%)	1,031 K (58%)
IMAP	143	4,202 K	473 K (11%)	1,759 K (42%)	1,781 K	223 K (13%)	1,022 K (57%)
HTTPS	443	34,727 K	5,975 K (17%)	11,444 K (33%)	17,490 K	1,749 K (10%)	3,931 K (22%)
SMTSPS	465	3,596 K	291 K (8%)	1,439 K (40%)	1,641 K	40 K (2%)	949 K (58%)
SMTP	587	3,507 K	423 K (12%)	1,464 K (42%)	1,657 K	133 K (8%)	986 K (59%)
IMAPS	993	4,315 K	853 K (20%)	1,835 K (43%)	1,909 K	260 K (14%)	1,119 K (59%)
POP3S	995	4,322 K	884 K (20%)	1,919 K (44%)	1,974 K	304 K (15%)	1,191 K (60%)
(Alexa Top 1M)	443	611 K	82 K (13%)	152 K (25%)	456 K	38 K (8%)	109 K (24%)

Table 4: **Hosts vulnerable to general DROWN.** We performed Internet-wide scans to measure the number of hosts supporting SSLv2 on several different protocols. A host is vulnerable to DROWN if its public key is exposed anywhere via SSLv2. Overall vulnerability to DROWN is much larger than support for SSLv2 due to widespread reuse of keys.

7 Measurements

We performed Internet-wide scans to analyze the number of systems vulnerable to DROWN. A host is directly vulnerable to general DROWN if it supports SSLv2. Similarly, a host is directly vulnerable to special DROWN if it supports SSLv2 and has the extra clear bug (which also implies the leaky export bug). These directly vulnerable hosts can be used as oracles to attack any other host with the same key. Hosts that do not support SSLv2 are still vulnerable to general or special DROWN if their RSA key pair is exposed by any general or special DROWN oracle, respectively. The oracles may be on an entirely different host or port. Additionally, any host serving a browser-trusted certificate is vulnerable to a special DROWN man-in-the-middle if any name on the certificate appears on any other certificate containing a key that is exposed by a special DROWN oracle.

We used ZMap [16] to perform full IPv4 scans on eight different ports during late January and February 2016. We examined port 443 (HTTPS), and common email ports 25 (SMTP with STARTTLS), 110 (POP3 with STARTTLS), 143 (IMAP with STARTTLS), 465 (SMTSPS), 587 (SMTP with STARTTLS), 993 (IMAPS), and 995 (POP3S). For each open port, we attempted three complete handshakes: one normal handshake with the highest available SSL/TLS version; one SSLv2 handshake requesting an export RC2 cipher suite; and one SSLv2 handshake with a non-export cipher and sixteen bytes of plaintext key material sent during key exchange, which we used to detect if a host has the extra clear bug.

We summarize our general DROWN results in Table 4. The fraction of SSL/TLS hosts that directly supported SSLv2 varied substantially across ports. 28% of SMTP servers on port 25 supported SSLv2, likely due to the opportunistic encryption model for email transit. Since SMTP fails-open to plaintext, many servers are config-

ured with support for the largest possible set of protocol versions and cipher suites, under the assumption that even bad or obsolete encryption is better than plaintext [9]. The other email ports ranged from 8% for SMTSPS to 20% for POP3S and IMAPS. We found 17% of all HTTPS servers, and 10% of those with a browser-trusted certificate, are directly vulnerable to general DROWN.

OpenSSL SSLv2 cipher suite selection bug. We discovered that OpenSSL servers do not respect the cipher suites advertised in the SSLv2 ServerHello message. That is, a malicious client can select an *arbitrary* cipher suite in the ClientMasterKey message, regardless of the contents of the ServerHello, and force the use of export cipher suites even if they are explicitly disabled in the server configuration. To fully detect SSLv2 oracles, we configured our scanner to ignore the ServerHello cipher list. The cipher selection bug helps explain the wide support for SSLv2—the protocol appeared disabled, but non-standard clients could still complete handshakes.

Widespread public key reuse. Reuse of RSA key material across hosts and certificates is widespread [21, 23]. Often this is benign: organizations may issue multiple TLS certificates for distinct domains with the same public key in order to simplify use of TLS acceleration hardware and load balancing. However, there is also evidence that system administrators may not entirely understand the role of the public key in certificates. For example, in the wake of the Heartbleed vulnerability, a substantial fraction of compromised certificates were reissued with the same public key [15]. The number of hosts vulnerable to DROWN rises significantly when we take RSA key reuse into account. For HTTPS, 17% of hosts are vulnerable to general DROWN because they support both TLS and SSLv2 on the HTTPS port, but 33% are vulnerable when considering RSA keys used by another service.

Protocol	Port	SSL/TLS	Any certificate		Trusted certificates		
			Special DROWN oracles	Vulnerable key	SSL/TLS	Vulnerable key	Vulnerable name
SMTP	25	3,357 K	855 K (25%)	896 K (27%)	1,083 K	305 K (28%)	398 K (37%)
POP3	110	4,193 K	397 K (9%)	946 K (23%)	1,787 K	485 K (27%)	674 K (38%)
IMAP	143	4,202 K	457 K (11%)	969 K (23%)	1,781 K	498 K (30%)	690 K (39%)
HTTPS	443	34,727 K	4,029 K (12%)	9,089 K (26%)	17,490 K	2,523 K (14%)	3,793 K (22%)
SMTPS	465	3,596 K	334 K (9%)	765 K (21%)	1,641 K	430 K (26%)	630 K (38%)
SMTP	587	3,507 K	345 K (10%)	792 K (23%)	1,657 K	482 K (29%)	667 K (40%)
IMAPS	993	4,315 K	892 K (21%)	1,073 K (25%)	1,909 K	602 K (32%)	792 K (42%)
POP3S	995	4,322 K	897 K (21%)	1,108 K (26%)	1,974 K	641 K (32%)	835 K (42%)
(Alexa Top 1M)	443	611 K	22 K (4%)	52 K (9%)	456 K	33 K (7%)	85 K (19%)

Table 5: **Hosts vulnerable to special DROWN.** A server is vulnerable to special DROWN if its key is exposed by a host with the CVE-2016-0703 bug. Since the attack is fast enough to enable man-in-the-middle attacks, a server is also vulnerable (to impersonation) if any name in its certificate is found in any trusted certificate with an exposed key.

Special DROWN. As shown in Table 5, 9.1 M HTTPS servers (26%) are vulnerable to special DROWN, as are 2.5 M HTTPS servers with browser-trusted certificates (14%). 66% as many HTTPS hosts are vulnerable to special DROWN as to general DROWN (70% for browser-trusted servers). While 2.7 M public keys are vulnerable to general DROWN, only 1.1 M are vulnerable to special DROWN (41% as many). Vulnerability among Alexa Top Million domains is also lower, with only 9% of domains vulnerable (7% for browser-trusted domains).

Since special DROWN enables active man-in-the-middle attacks, any host serving a browser-trusted certificate with at least one name that appears on any certificate with an RSA key exposed by a special DROWN oracle is vulnerable to an impersonation attack. Extending our search to account for certificates with shared names, we find that 3.8 M (22%) hosts with browser-trusted certificates are vulnerable to man-in-the-middle attacks, as well as 19% of the browser-trusted domains in the Alexa Top Million.

8 Related work

TLS has had a long history of implementation flaws and protocol attacks [2, 3, 7, 14, 15, 35, 38]. We discuss relevant Bleichenbacher and cross-protocol attacks below.

Bleichenbacher’s attack. Bleichenbacher’s adaptive chosen ciphertext attack against SSL was first published in 1998 [8]. Several works have adapted his attack to different scenarios [4, 25, 29]. The TLS standard explicitly introduces countermeasures against the attack [13], but several modern implementations have been discovered to be vulnerable to timing-attack variants in recent years [34, 42]. These side-channel attacks are implementation failures and only apply when the attacker is co-located with the victim.

Cross-protocol attacks. Jager et al. [26] showed that a cross-protocol Bleichenbacher RSA padding oracle attack is possible against the proposed TLS 1.3 standard, in spite of the fact that TLS 1.3 does not include RSA key exchange, if server implementations use the same certificate for previous versions of TLS and TLS 1.3. Wagner and Schneier [41] developed a cross-cipher suite attack for SSLv3, in which an attacker could reuse a signed server key exchange message in a later exchange with a different cipher suite. Mavrogiannopoulos et al. [32] developed a cross-cipher suite attack allowing an attacker to use elliptic curve Diffie-Hellman as prime field Diffie-Hellman.

Attacks on export-grade cryptography. Recently, the FREAK [5] and Logjam [1] attacks allowed an active attacker to downgrade a connection to export-grade RSA and Diffie-Hellman, respectively. DROWN exploits export-grade symmetric ciphers, completing the export-grade cryptography attack trifecta.

9 Discussion

9.1 Implications for modern protocols

Although the protocol flaws in SSLv2 enabling DROWN are not present in recent TLS versions, many modern protocols meet a subset of the requirements to be vulnerable to a DROWN-style attack. For example:

1. RSA key exchange. TLS 1.2 [13] allows this.
2. Reuse of server-side nonce by the client. QUIC [10] allows this.
3. Server sends a message encrypted with the derived key before the client. QUIC, TLS 1.3 [37], and TLS False Start [30] do this.
4. Deterministic cipher parameters are generated from the premaster secret and nonces. This is the case for all TLS stream ciphers and TLS 1.0 block ciphers.

DROWN has a natural adaptation when all three properties are present. The attacker exposes a Bleichenbacher oracle by connecting to the server twice with the identical RSA ciphertexts and server-side nonces. If the RSA ciphertext is PKCS conformant, the server will respond with identical messages across both connections; otherwise they will differ.

9.2 Lessons for key reuse

DROWN illustrates the cryptographic principle that keys should be single use. Often, this principle is primarily applied to keys that are used to both sign and decrypt, but DROWN illustrates that using keys *for different protocol versions* can also be a serious security risk. Unfortunately, there is no widely supported way to pin X.509 certificates to specific protocols. While using per-protocol certificates may help defend against passive attacks, an active attacker could still leverage any certificate with a matching name.

9.3 Harms from obsolete cryptography

Recent years have seen a significant number of serious attacks exploiting outdated and obsolete cryptography. Many protocols and cryptographic primitives that were demonstrated to be weak decades ago are surprisingly common in real-world systems.

DROWN exploits a modification of an 18-year-old attack against a combination of protocols and ciphers that have long been superseded by better options: the SSLv2 protocol, export cipher suites, and PKCS #1 v1.5 RSA padding. In fact, support for RSA as a key exchange method, including the use of PKCS #1 v1.5, is mandatory even for TLS 1.2. The attack is made more severe by implementation flaws in rarely used code.

Our work serves as yet another reminder of the importance of removing deprecated technologies before they become exploitable vulnerabilities. In response to many of the vulnerabilities listed above, browser vendors have been aggressively warning end users when TLS connections are negotiated with unsafe cryptographic parameters, including SHA-1 certificates, small RSA and Diffie-Hellman parameters, and SSLv3 connections. This process is currently happening in a piecemeal fashion, primitive by primitive. Vendors and developers rightly prioritize usability and backward compatibility in standards, and are willing to sacrifice these only for practical attacks. This approach works less well for cryptographic vulnerabilities, where the first sign of a weakness, while far from being practically exploitable, can signal trouble in the future. Communication issues between academic researchers and vendors and developers have been voiced by many in the community, including Green [18] and Jager et al. [24].

The long-term solution is to proactively remove these obsolete technologies. There is movement towards this

already: TLS 1.3 has entirely removed RSA key exchange and has restricted Diffie-Hellman key exchange to a few groups large enough to withstand cryptanalytic attacks long in the future. The CA/Browser forum will remove support for SHA-1 certificates this year. Resources such as the SSL Labs SSL Reports have gathered information about best practices and vulnerabilities in one place, in order to encourage administrators to make the best choices.

9.4 Harms from weakening cryptography

Export-grade cipher suites for TLS deliberately weakened three primitives to the point that they are now broken even to enthusiastic amateurs: 512-bit RSA key exchange, 512-bit Diffie-Hellman key exchange, and 40-bit symmetric encryption. All three deliberately weakened primitives have been cornerstones of high-profile attacks: FREAK exploits export RSA, Logjam exploits export Diffie-Hellman, and now DROWN exploits export symmetric encryption.

Like FREAK and Logjam, our results illustrate the continued harm that a legacy of deliberately weakened export-grade cryptography inflicts on the security of modern systems, even decades after the regulations influencing the original design were lifted. The attacks described in this paper are fully feasible against export cipher suites today. The technical debt induced by cryptographic “front doors” has left implementations vulnerable for decades. With the slow rate at which obsolete protocols and primitives fade away, we can expect some fraction of hosts to remain vulnerable for years to come.

Acknowledgements

The authors thank team Hashcat for making their GPUs available for the execution of the attack, Ralph Holz for providing early scan data, Adam Langley for insights about QUIC, Graham Steel for insights about TLS False Start, the OpenSSL team for their help with disclosure, Ivan Ristic for comments on session resumption in a BEAST-styled attack, and Tibor Jager and Christian Mainka for further helpful comments. We thank the exceptional sysadmins at the University of Michigan for their help and support throughout this project, including Chris Brenner, Kevin Cheek, Laura Fink, Dan Maletta, Jeff Richardson, Donald Welch, Don Winsor, and others from ITS, CAEN, and DCO.

This material is based upon work supported by the U.S. National Science Foundation under Grants No. CNS-1345254, CNS-1408734, CNS-1409505, CNS-1505799, CNS-1513671, and CNS-1518888, an AWS Research Education grant, a scholarship from the Israeli Ministry of Science, Technology and Space, a grant from the Blavatnik Interdisciplinary Cyber Research Center (ICRC) at Tel Aviv University, a gift from Cisco, and an Alfred P. Sloan Foundation research fellowship.

References

- [1] ADRIAN, D., BHARGAVAN, K., DURUMERIC, Z., GAUDRY, P., GREEN, M., HALDERMAN, J. A., HENINGER, N., SPRINGALL, D., THOMÉ, E., VALENTE, L., VANDERSLOOT, B., WUSTROW, E., ZANELLA-BÉGUELIN, S., AND ZIMMERMANN, P. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security* (Oct. 2015).
- [2] AL FARDAN, N. J., AND PATERSON, K. G. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 526–540.
- [3] ALFARDAN, N. J., BERNSTEIN, D. J., PATERSON, K. G., POETTERING, B., AND SCHULDT, J. C. On the security of RC4 in TLS. In *22nd USENIX Security Symposium* (2013), pp. 305–320.
- [4] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., SIMIONATO, L., STEEL, G., AND TSAY, J.-K. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology—CRYPTO 2012*. Springer, 2012, pp. 608–625.
- [5] BEURDOUCHE, B., BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., KOHLWEISS, M., PIRONTI, A., STRUB, P.-Y., AND ZINZINDOHOUE, J. K. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy* (2015).
- [6] BHARGAVAN, K., LAVAUD, A. D., FOURNET, C., PIRONTI, A., AND STRUB, P. Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 98–113.
- [7] BHARGAVAN, K., AND LEURENT, G. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *Network and Distributed System Security Symposium* (Feb. 2016).
- [8] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98*, vol. 1462 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998.
- [9] BREYHA, W., DURVAUX, D., DUSSA, T., KAPLAN, L. A., MENDEL, F., MOCK, C., KOSCHUCH, M., KRIEGISCH, A., PÖSCHL, U., SABET, R., SAN, B., SCHLATTERBECK, R., SCHRECK, T., WÜRSTLEIN, A., ZAUNER, A., AND ZAWODSKY, P. Better crypto – applied crypto hardening, 2016. Available at <https://bettercrypto.org/static/applied-crypto-hardening.pdf>.
- [10] CHANG, W.-T., AND LANGLEY, A. QUIC crypto, 2014. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit?pli=1.
- [11] CVE-2015-0293. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0293>.
- [12] DE RUITER, J., AND POLL, E. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium* (Washington, D.C., Aug. 2015), USENIX Association.
- [13] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard, Aug. 2008. Updated by RFCs 5746, 5878.
- [14] DUONG, T., AND RIZZO, J. Here come the xor ninjas, 2011. http://netifera.com/research/beast/beast_DRAFT_0621.pdf.
- [15] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., AND PAXSON, V. The matter of Heartbleed. In *14th Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 475–488.
- [16] DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. ZMap: Fast Internet-wide scanning and its security applications. In *22nd USENIX Security Symposium* (Aug. 2013).
- [17] FREIER, A., KARLTON, P., AND KOCHER, P. The secure sockets layer (SSL) protocol version 3.0. RFC 6101, 2011.
- [18] GREEN, M. Secure protocols in a hostile world. In *CHES 2015* (Aug. 2015). <https://isi.jhu.edu/~mgreen/CHESPDF.pdf>.
- [19] HAMILTON, R. QUIC discovery. <https://docs.google.com/document/d/1i4m7DbrWGgXafHxwl8SwIusY2ELUe8WX258xt2LFxPM/edit#>.
- [20] Hashcat. <http://hashcat.net>.
- [21] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium* (Aug. 2012).
- [22] HICKMAN, K., AND ELGAMAL, T. The SSL protocol, 1995. <https://tools.ietf.org/html/draft-hickman-netscape-ssl-00>.
- [23] HOLZ, R., AMANN, J., MEHANI, O., WACHS, M., AND KAAFAR, M. A. TLS in the wild: An Internet-wide analysis of TLS-based protocols for electronic communication. In *Network and Distributed System Security Symposium* (Geneva, Switzerland, Feb. 2016), S. Capkun, Ed., Internet Society.
- [24] JAGER, T., PATERSON, K. G., AND SOMOROVSKY, J. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *Network and Distributed System Security Symposium* (2013).
- [25] JAGER, T., SCHINZEL, S., AND SOMOROVSKY, J. Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. In *17th European Symposium on Research in Computer Security* (Berlin, Heidelberg, 2012), Springer Berlin Heidelberg, pp. 752–769.
- [26] JAGER, T., SCHWENK, J., AND SOMOROVSKY, J. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In *22nd ACM Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1185–1196.
- [27] KALISKI, B. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational), Mar. 1998. Obsoleted by RFC 2437.
- [28] KÄSPER, E. Fix reachable assert in SSLv2 servers. OpenSSL patch, Mar. 2015. <https://github.com/openssl/openssl/commit/86f8fb0e344d62454f8daf3e15236b2b59210756>.
- [29] KLIMA, V., POKORNÝ, O., AND ROSA, T. Attacking RSA-based sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems—CHES 2003*. Springer, 2003, pp. 426–440.
- [30] LANGLEY, A., MODADUGU, N., AND MOELLER, B. Transport layer security (TLS) false start. *draft-bmoeller-tls-falsestart-00*, June 2 (2010).
- [31] LENSTRA, A. K., LENSTRA, H. W., AND LOVÁSZ, L. Factoring polynomials with rational coefficients. *Mathematische Annalen* 261 (1982), 515–534. 10.1007/BF01457454.
- [32] MAVROGIANNOPoulos, N., VERCAUTEREN, F., VELICHKOV, V., AND PRENEEL, B. A cross-protocol attack on the TLS protocol. In *19th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 62–72.
- [33] MEYER, C., AND SCHWENK, J. SoK: Lessons learned from SSL/TLS attacks. In *14th International Workshop on Information Security Applications* (Berlin, Heidelberg, Aug. 2013), WISA 2013, Springer-Verlag.
- [34] MEYER, C., SOMOROVSKY, J., WEISS, E., SCHWENK, J., SCHINZEL, S., AND TEWS, E. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium*. USENIX Association, San Diego, CA, Aug. 2014, pp. 733–748.

- [35] MÖLLER, B., DUONG, T., AND KOTOWICZ, K. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.
- [36] OPENSSL. Change log. <https://www.openssl.org/news/changelog.html#x0>.
- [37] RESCORLA, E., ET AL. The transport layer security (TLS) protocol version 1.3, draft.
- [38] RIZZO, J., AND DUONG, T. The CRIME attack. EKOParty Security Conference, 2012.
- [39] ROSKIND, J. QUIC design document, 2013. https://docs.google.com/a/chromium.org/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34.
- [40] TURNER, S., AND POLK, T. Prohibiting secure sockets layer (SSL) version 2.0. RFC 6176 (Informational), Apr. 2011.
- [41] WAGNER, D., AND SCHNEIER, B. Analysis of the SSL 3.0 protocol. In *2nd USENIX Workshop on Electronic Commerce* (1996).
- [42] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *21st ACM Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 990–1003.

A Adaptations to Bleichenbacher’s attack

A.1 Success probability of fractions

For a given fraction u/t , the success probability with a randomly chosen TLS conformant ciphertext can be computed as follows. Let m_0 be a random TLS conformant message, $m_1 = m_0 \cdot u/t$, and let ℓ_k be the expected length of the unpadded message. For $s = u/t \bmod N$ where u and t are coprime, m_1 will be SSLv2 conformant if the following conditions all hold:

1. m_0 is divisible by t . For a randomly generated m_0 , this condition holds with probability $1/t$.
2. $m_1[1] = 0$ and $m_1[2] = 2$, or the integer $m \cdot u/t \in [2B, 3B]$. For a randomly generated m_0 divisible by t , this condition holds with probability

$$P = \begin{cases} 3 - 2 \cdot t/u & \text{for } 2/3 < u/t < 1 \\ 3 \cdot t/u - 2 & \text{for } 1 < u/t < 3/2 \\ 0 & \text{otherwise} \end{cases}$$

3. $\forall i \in [3, \ell_m - (\ell_k + 1)]$, $m_1[i] \neq 0$, or all bytes between the first two bytes and the $(k + 1)$ least significant bytes are non-zero. This condition holds with probability $(1 - 1/256)^{\ell_m - (\ell_k + 3)}$.
4. $m_1[\ell_m - \ell_k] = 0$: the $(\ell_k + 1)$ st least significant byte is 0. This condition holds with probability $1/256$.

Using the above formulas for $u/t = 7/8$, the overall probability of success is $P = 1/8 \cdot 0.71 \cdot 0.37 \cdot 1/256 = 1/7,774$; thus the attacker expects to find an SSLv2 conformant ciphertext after testing 7,774 randomly chosen TLS conformant ciphertexts. The attacker can decrease the number of TLS conformant ciphertexts needed by multiplying each candidate ciphertext by several fractions.

Note that testing random s values until $c_1 = c_0 \cdot s^e \bmod N$ is SSLv2 conformant yields a success probability of $P_{rnd} \approx (1/256)^3 * (255/256)^{249} \approx 2^{-25}$.

A.2 Optimizing the chosen set of fractions

In order to deduce the validity of a single ciphertext, the attacker would have to perform a non-trivial brute-force search over all 5 byte `master_key` values. This translates into 2^{40} encryption operations.

The search space can be reduced by an additional optimization, relying on the fractional multipliers used in the first step. If the attacker uses $u/t = 8/7$ to compute a new SSLv2 conformant candidate, and m_0 is indeed divisible by $t = 7$, then the new candidate message $m_1 = m_0/t \cdot u$ is divisible by $u = 8$, and the last three bits of m_1 (and thus mk_{secret}) are zero. This allows reducing the searched `master_key` space by selecting specific fractions.

More generally, for an integer u , the largest power of 2 by which u is divisible is denoted by $v_2(u)$, and multiplying by a fraction u/t reduces the search space by a factor of $v_2(u)$. With this observation, the trade-off between the 3 metrics: the required number of intercepted ciphertexts, the required number of queries, and the required number of encryption attempts, becomes non-trivial to analyze.

Therefore, we have resorted to using simulations when evaluating the performance metrics for sets of fractions. The probability that multiplying a ciphertext by any fraction out of a given set of fractions results in an SSLv2 conformant message is difficult to compute, since the events are in fact inter-dependent: If $m \cdot 16/15$ is conforming, then m is divisible by 5, greatly increasing the probability that $m \cdot 4/5$ is also conforming. However, it is easy to perform a Monte Carlo simulation, where we randomly generate ciphertexts, and measure the probability that any fraction out of a given set produces a conforming message. The expected required number of intercepted ciphertexts is the inverse of that probability.

Formally, if we denote the set of fractions as F , and the event that a message m is conforming as $C(m)$, we perform a Monte Carlo estimation of the probability $P_F = P(\exists f \in F : C(m \cdot f))$, and the expected number of required intercepted ciphertexts equals $1/P_F$. The required number of oracle queries is simply $1/P_F \cdot |F|$. Accordingly, the required number of server connections is $2 \cdot 1/P_F \cdot |F|$, since each oracle query requires two server connections. And as for the required number of encryption attempts, if we denote this number when querying with a given fraction $f = u/t$ as E_f , then $E_f = E_{u/t} = 2^{40-v_2(u)}$. We further define the required encryption attempts when testing a ciphertext with a given set of fraction F as $E_F = \sum_{f \in F} E_f$. Then the required number of encryption attempts in Phase 1 for a given set of fractions is $(1/P_F) \cdot E_F$.

We can now give precise figures for the expected number of required intercepted ciphertexts, connections to the targeted server, and encryption attempts. The results presented in Table 1 were obtained using the above approach with one billion random ciphertexts per fraction set F .

A.3 Rotation and multiplier speedups

For a randomly chosen s , the probability that the two most significant bytes are 0x00 02 is 2^{-16} ; for a 2028-bit modulus N the probability that the next $\ell_m - \ell_k - 3$ bytes of m_2 are all nonzero is about 0.37 as in the previous section, and the probability that the $\ell_k + 1$ least significant delimiter byte is 0x00 is 1/256. Thus a randomly chosen s will work with probability $2^{-25.4}$ and the attacker expects to try $2^{25.4}$ values for s before succeeding.

However, since the attacker has already learned $\ell_k + 3$ most significant bytes of $m_1 \cdot R^{-1} \pmod{N}$, for $\ell_k \geq 4$ and $s < 2^{30}$ they do not need to query the oracle to learn if the two most significant bytes are SSLv2 conformant; they can compute this themselves from their knowledge of $\tilde{m}_1 \cdot R^{-1}$. They iterate through values of s , test that the top two bytes of $\tilde{m}_1 \cdot R^{-1} \pmod{N}$ are 0x00 02, and only query the oracle for s values that satisfy this test. Therefore, for a 2048-bit modulus they expect to test 2^{16} values offline per oracle query. The probability that a query is conformant is then $P = (1/256) * (255/256)^{249} \approx 1/678$, so they expect to perform 678 oracle queries before finding a fully SSLv2 conformant ciphertext $c_2 = (s \cdot R^{-1})^e c_1 \pmod{N}$.

We can speed up the brute force testing of 2^{16} values of s using algebraic lattices. We are searching for values of s satisfying $\tilde{m}_1 R^{-1} s < 3B \pmod{N}$, or given an offset s_0 we would like to find solutions x and z to the equation $\tilde{m}_1 R^{-1} (s_0 + x) = 2B + z \pmod{N}$ where $|x| < 2^{16}$ and $|z| < B$. Let $X = 2^{15}$. We can construct the lattice basis

$$L = \begin{bmatrix} -B & X\tilde{m}_1 R^{-1} & \tilde{m}_1 R^{-1} s_0 + B \\ 0 & XN & 0 \\ 0 & 0 & N \end{bmatrix}$$

We then run the LLL algorithm [31] on L to obtain a reduced lattice basis V containing vectors v_1, v_2, v_3 . We then construct the linear equations $f_1(x, z) = v_{1,1}/B \cdot z + v_{1,2}/X \cdot x + v_{1,3} = 0$ and $f_2(x, z) = v_{2,1}/B \cdot z + v_{2,2}/X \cdot x + v_{2,3} = 0$ and solve the system of equations to find a candidate integer solution $x = \tilde{s}$. We then test $s = \tilde{s} + s_0$ as our candidate solution in this range.

$\det L = XZN^2$ and $\dim L = 3$, thus we expect the vectors v_i in V to have length approximately $|v_i| \approx (XZN^2)^{1/3}$. We will succeed if $|v_i| < N$, or in other words $XZ < N$. $N \approx 2^{8\ell_m}$, so we expect to find short enough vectors. This approach works well in practice and is significantly faster than iterating through 2^{16} possible values of \tilde{s} for each query.

In summary, given an SSLv2 conformant ciphertext $c_1 = m_1^e \pmod{N}$, we can efficiently generate an SSLv2 conformant ciphertext $c_2 = m_2^e \pmod{N}$ where $m_2 = s \cdot m_1 \cdot R^{-1} \pmod{N}$ and we know several most significant bytes of m_2 , using only a few hundred oracle queries in expectation. We can iterate this process as many times as we like to continue generating SSLv2 conformant ciphertexts c_i for which we know increasing numbers of most

significant bytes, and which have a known multiplicative relationship to our original message c_0 .

A.4 Rotations in the general DROWN attack

After the first phase, we have learned an SSLv2 conformant ciphertext c_1 , and we wish to shift known plaintext bytes from least to most significant bits. Since we learn the least significant 6 bytes of plaintext of m_1 from a successful oracle $\mathcal{O}_{\text{SSLv2-export}}$ query, we could use a shift of 2^{-48} to transfer 48 bits of known plaintext to the most significant bits of a new ciphertext. However, we perform a slight optimization here, to reduce the number of encryption attempts. We instead use a shift of 2^{-40} , so that the least significant byte of $m_1 \cdot 2^{-40}$ and $\tilde{m}_1 \cdot 2^{-40}$ will be known. This means that we can compute the least significant byte of $m_1 \cdot 2^{-40} \cdot s \pmod{N}$, so oracle queries now only require 2^{32} encryption attempts each. This brings the total expected number of encryption attempts for each shift to $2^{32} * 678 \approx 2^{41}$.

We perform two such plaintext shifts in order to obtain an SSLv2 conformant message, m_3 that resides in a narrow interval of length at most $2^{8\ell-66}$. We can then obtain a multiplier s_3 such that $m_3 \cdot s_3$ is also SSLv2 conformant. Since m_3 lies in an interval of length at most $2^{8\ell-66}$, with high probability for any $s_3 < 2^{30}$, $m_3 \cdot s_3$ lies in an interval of length at most $2^{8\ell_m-36} < B$, so we know the two most significant bytes of $m_3 \cdot s_3$. Furthermore, we know the value of the 6 least significant bytes after multiplication. We therefore test possible values of s_3 , and for values such that $m_3 \cdot s_3 \in [2B, 3B]$, and $(m_3 \cdot s_3)[\ell_m - 5] = 0$, we query the oracle with $c_3 \cdot s_3^e \pmod{N}$. The only condition for PKCS conformance which we haven't verified before querying the oracle is the requirement of non-zero padding, which holds with probability 0.37.

In summary, after roughly $1/0.37 = 2.72$ queries we expect a positive response from the oracle. Since we know the value of the 6 least significant bytes after multiplication, this phase does not require performing an exhaustive search. If the message is SSLv2 conformant after multiplication, we know the symmetric key, and can test whether it correctly decrypts the `ServerVerify` message.

A.5 Adapted Bleichenbacher iteration

After we have bootstrapped the attack using rotations, the original algorithm proposed by Bleichenbacher can be applied with minimal modifications.

The original step obtains a message that starts with the required 0x00 02 bytes once in roughly every two queries on average, and requires the number of queries to be roughly $16\ell_m$. Since we know the value of the 6 least significant bytes after multiplying by any integer, we can only query the oracle for multipliers that result in a zero 6th least significant byte, and again an exhaustive search over keys is not required. However, we cannot ensure

that the padding is non-zero when querying, which again holds with probability 0.37. Therefore, for a 2048-bit modulus, the overall expected number of queries for this phase is roughly $2048 * 2 / 0.37 = 11,070$.

A.6 Special DROWN MITM performance

For the first step, the probability that the three padding bytes are correct remains unchanged. The probability that all the intermediate padding bytes are non-zero is now slightly higher, $P_1 = (1 - 1/256)^{229} = 0.41$, yielding an overall maximal success probability $P = 0.1 \cdot 0.41 \cdot \frac{1}{256} = 1/6,244$ per oracle query. Since the attacker now only needs to connect to the server once per oracle query, the expected number of connections in this step is the same, 6,243. Phase 1 now yields a message with 3 known padding bytes and 24 known plaintext bytes.

For the remaining rotation steps, each rotation requires an expected 630 oracle queries. The attacker could now complete the original Bleichenbacher attack by performing 11,000 sequential queries in the final phase. However, with this more powerful oracle it is more efficient to apply a rotation 10 more times to recover the remaining plaintext bits. The number of queries required in this phase is now $10 \cdot 256 / 0.41 \approx 6,300$, and the queries for each of the 10 steps can be executed in parallel.

Using multiple queries per fraction. For the $\mathcal{O}_{\text{SSLv2-extra-clear}}$ oracle, the attacker can increase their chances of success by querying the server multiple times per ciphertext and fraction, using different cipher suites with different key lengths. They can negotiate DES and hope the 9th least significant byte is zero, then negotiate 128-bit RC4 and hope the 17th least significant byte is zero, then negotiate 3DES and hope the 25th least significant is zero. All three queries also require the intermediate padding bytes to be non-zero. This technique triples the success probability for a given pair of (ciphertext, fraction), at a cost of triple the queries. Its primary benefit is that fractions with smaller denominators (and thus higher probabilities of success) are now even more likely to succeed.

For a random ciphertext, when choosing 70 fractions, the probability of the first zero delimiter byte being in one of these three positions is 0.01. Hence, the attacker can use only 100 recorded ciphertexts, and expect to use $100 * 70 * 3 = 21,000$ oracle queries. For the Extra Clear oracle, each query requires one SSLv2 connection to the server. After obtaining the first positive response from the oracle, the attacker proceeds to phase 2 using 3DES.

A.7 Special DROWN with combined oracles

Using the Leaky Export oracle, the probability that a fraction u/t will result in a positive response is $P = P_0 * P_3$, where the formula for computing $P_0 = P((m \cdot u/t)[1, 2] = 00||02)$ is provided in Appendix A.1, and P_3 is, for a

2048-bit modulus:

$$\begin{aligned} P_3 &= P(0x00 \notin \{m_3, \dots, m_{10}\} \wedge \\ &\quad 0x00 \in \{m_{11}, \dots, m_\ell\}) \quad (1) \\ &= (1 - 1/256)^8 * (1 - (1 - 1/256)^{246}) = 0.60 \end{aligned}$$

Phase 1. Our goal for this phase is to obtain a divisor t as large as possible, such that $t|m$. We generate a list of fractions, sorted in descending order of the probability of resulting in a positive response from $\mathcal{O}_{\text{SSLv2-export-leaky}}$. For a given ciphertext c , we then query with the 50 fractions in the list with the highest probability, until we obtain a first positive response for a fraction u_0/t_0 . We can now deduce that $t_0|m$. We then generate a list of fractions u/t where t is a multiple of t_0 , sort them again by success probability, and again query with the 50 most probable fractions, until a positive answer is obtained, or the list is exhausted. If a positive answer is obtained, we iteratively re-apply this process, until the list is exhausted, resulting in a final fraction u^*/t^* .

Phase 2. We then query with all fractions denominated by t^* , and hope the ciphertext decrypts to a plaintext of one of seven possible lengths: $\{2, 3, 4, 5, 8, 16, 24\}$. Assuming that this is the case, we learn at least three least significant bytes, which allows us to use the shifting technique in order to continue the attack. Detecting plaintext lengths 8, 16 and 24 can be accomplished using three Extra Clear oracle queries, employing DES, 128-bit RC4 and 3DES, respectively, as the chosen cipher suite. Detecting plaintext lengths 2, 3, 4 and 5 can be accomplished by using a single Leaky Export oracle query, which requires at most 2^{41} offline computation. In fact, the optimization over the key search space described in Section 3.2.1 is applicable here and can slightly reduce the required computation. Therefore, by initiating four SSLv2 connections and performing at most 2^{41} offline work, the attacker can test for ciphertexts which decrypt to one of these seven lengths.

In practice, choosing 50 fractions per iteration as described above results in a success probability of 0.066 for a single ciphertext. Hence, the expected number of required ciphertexts is merely $1 / 0.066 = 15$. The expected number of fractions per ciphertext for phase 1 is 60, as in most cases phase 1 consists of just a few successful iterations. Since each fraction requires a single query to $\mathcal{O}_{\text{SSLv2-export-leaky}}$, the overall number of queries for this stage is $15 * 60 = 900$, and the required offline computation is at most $900 * 2^{41} \approx 2^{51}$, which is similar to general DROWN. For a 2048-bit RSA modulus, the expected number of queries for phase 2 is 16. Each query consists of three queries to $\mathcal{O}_{\text{SSLv2-extra-clear}}$ and one query to $\mathcal{O}_{\text{SSLv2-export-leaky}}$, which requires at most 2^{41} computation. Therefore in expectancy the attacker has to perform 2^{45} offline computation for phase 2.

All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption

Yupeng Zhang* Jonathan Katz† Charalampos Papamanthou*

Abstract

The goal of *searchable encryption* (SE) is to enable a client to execute searches over encrypted files stored on an untrusted server while ensuring some measure of privacy for both the encrypted files and the search queries. Most recent research has focused on developing efficient SE schemes at the expense of allowing some small, well-characterized “(information) leakage” to the server about the files and/or the queries. The practical impact of this leakage, however, remains unclear.

We thoroughly study *file-injection attacks*—in which the server sends files to the client that the client then encrypts and stores—on the query privacy of single-keyword and conjunctive SE schemes. We show such attacks can reveal the client’s queries in their entirety using very few injected files, even for SE schemes having low leakage. We also demonstrate that natural countermeasures for preventing file-injection attacks can be easily circumvented. Our attacks outperform prior work significantly in terms of their effectiveness as well as in terms of their assumptions about the attacker’s prior knowledge.

1 Introduction

The goal of *searchable encryption* (SE) is to enable a client to perform keyword searches over encrypted files stored on an untrusted server while still guaranteeing some measure of privacy for both the files themselves as well as the client’s queries. In principle, solutions that leak no information to the server can be constructed based on powerful techniques such as secure

*Department of Electrical and Computer Engineering, University of Maryland. Research supported in part by NSF awards #1514261 and #1526950, by a Google Faculty Research Award, and by Yahoo! Labs through the Faculty Research Engagement Program (FREP). Email: {zhangyp, cpap}@umd.edu.

†Department of Computer Science, University of Maryland. Research supported in part by NSF awards #1223623 and #1514261. Email: jkatz@cs.umd.edu.

two-party computation, fully-homomorphic encryption, and/or oblivious RAM. Such systems, however, would be prohibitively expensive and completely impractical [15].

In light of the above, researchers have focused on the development of novel SE schemes that are much more efficient, at the expense of allowing some information to “leak” to the server [19, 9, 8, 11, 6, 16, 12, 20, 13, 5]. The situation is summarized, e.g., by Cash et al. [6]:

The premise of [our] work is that in order to provide truly practical SSE solutions one needs to accept a certain level of leakage; therefore, the aim is to achieve an acceptable balance between leakage and performance.

The question then becomes: what sort of leakage is acceptable? Roughly speaking, and focusing on single-keyword search for simplicity, current state-of-the-art schemes leak mainly two things: the *query pattern* (i.e., when a query is repeated) and the *file-access pattern* (namely, which files are returned in response to each query); these are collectively called *L1 leakage* in [4]. The prevailing argument is that L1 leakage is inconsequential in practice, and so represents a reasonable sacrifice for obtaining an efficient SE scheme.

In truth, the ramifications of different types of leakage are poorly understood; indeed, characterizing the real-world consequences of the leakage of existing SE schemes was highlighted as an important open question in [6]. Recently, several groups have shown that even seemingly minor leakage can be exploited to learn sensitive information, especially if the attacker has significant prior knowledge about the client’s files or the keywords they contain. Islam et al. [10] (IKK12), who initiated this line of work, showed that if the server knows (almost) all the contents of the client’s files, then it can determine the client’s queries from L1 leakage. Cash et al. [4] (CGPR15) gave an improved attack that works for larger keyword universes while assuming (slightly) less knowledge about the files of the client. They also ex-

plored the effects of even greater leakage, and showed how query-recovery attacks could serve as a springboard for learning further information about the client’s files.

A different attack for query recovery was given by Liu et al. [14]. The attack assumes a known distribution on the keywords being searched by the client, and works only after the client issues a large number of queries.

1.1 Our Contributions

In this paper, we further investigate the consequences of leakage in SE schemes through the lens of *file-injection attacks*. In such attacks, the server sends files of its choice to the client, who then encrypts and uploads them as dictated by the SE scheme. This attack was introduced by Cash et al. [4], who called it a *known-document attack*. As argued by those authors, it would be quite easy to carry out such attacks: for example, if a client is using an SE scheme for searching email (e.g., Pmail [2]), with incoming emails processed automatically, then the server can inject files by simply sending email to the client (from a spoofed email address, if it wishes to avoid suspicion). We stress that the server otherwise behaves entirely in an “honest-but-curious” fashion.

We show that file-injection attacks are *devastating* for query privacy: that is, a server can learn a very high fraction of the keywords searched by the client, by injecting a relatively small number of files. Compared to prior work [10, 4], our attacks are both more effective in terms of the fraction of queries recovered and far less demanding in terms of the prior information the server knows. Our attacks differ in that the server must inject files, but as argued above this is easy to carry out in practice.

We consider both adaptive and non-adaptive attacks, where adaptivity refers (in part) to whether the server injects files before or after the client’s query is made. In particular, a non-adaptive attack injects files that can be used to break all future queries; An adaptive attack crafts the injected files using leakage of previously-observed queries. Our adaptive attacks are more effective, but assume the SE scheme does not satisfy *forward privacy* [7, 20]. (Forward privacy means that the server cannot tell if a newly inserted file matches previous search queries. With the exception of [7, 20], however, all efficient SE schemes supporting updates do not have forward privacy.) Our work thus highlights the importance of forward privacy in any real-world deployment.

1.2 Organization of the paper

We begin by showing a simple, binary-search attack that allows the server to learn 100% of the client’s queries with *no* prior knowledge about the client’s files. We then propose an easy countermeasure: limiting the number of

keywords that are indexed per file. (We show that this idea is viable insofar as it has limited effect on the utility of searchable encryption.) However, our attacks can be suitably modified to defeat this countermeasure, either using a larger number of injected files (but still no prior knowledge about the client’s files) or based on limited knowledge—as low as 10%—of the client’s files. Our attacks still outperform prior work [10, 4], having a significantly higher recovery rate and requiring a lower fraction of the client’s files to be known.

We additionally investigate the effectiveness of padding files with random keywords (suggested in [10, 4]) as another countermeasure against our attacks. We show that the performance of our attacks degrades only slightly when such padding is used, in contrast to prior attacks that fail completely.

Finally, we initiate a study of the implications of leakage on *conjunctive* queries, and show how to extend our attacks to this setting. Our attacks work against SE schemes having “ideal” leakage, but are even more effective against the scheme of Cash et al. [6] (the most efficient SE scheme allowing conjunctive queries), which suffers from larger leakage.

2 Background

For the purposes of this paper, only minimal background about searchable encryption (SE) is needed. At a high level, an SE scheme allows a client to store encrypted versions of its files on a server, such that at a later point in time the client can retrieve all files containing a certain keyword (or collection of keywords). We assume a set of keywords $K = \{k_0, k_1, \dots\}$ known to an attacker, and for simplicity view a file as an unordered set of keywords. (Although the order and multiplicity of the keywords matter, and a file may contain non-keywords as well, these details are irrelevant for our purposes.)

We assume an SE scheme in which searching for some keyword k is done via the following process (all efficient SE schemes work in this way): first, the client deterministically computes a *token* t corresponding to k and sends t to the server; using t , the server then computes and sends back the *file identifiers* of all files containing keyword k . (These file identifiers need not be “actual” filenames; they can instead simply be pointers to the appropriate encrypted files residing at the server.) The client then downloads the appropriate files.

Because the token is generated deterministically from the keyword, the server can tell when queries repeat and thus learn the *query pattern*; the returned file identifiers reveal the *file-access pattern*. Our attacks rely only on knowledge of the file-access pattern, though we additionally assume that the server can identify when a specific file identifier corresponds to some particular file injected

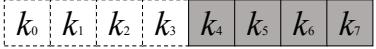
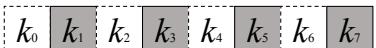
File 1:		0
File 2:		1
File 3:		0

Figure 1: An example of the binary-search attack with $|K| = 8$. Each file injected by the attacker contains 4 keywords, which are shaded in the figure. If file 2 is returned in response to some token, but files 1 and 3 are not, the keyword corresponding to that token is k_2 .

by the server. (The same assumption is made by Cash et al. [4].) This is reasonable to assume, even if file identifiers are chosen randomly by the client, for several reasons: (1) the server can identify the file returned based on its length (even if padding is used to mitigate this, it is impractical to pad every file to the maximum file length); (2) in SE schemes supporting updates, the server can inject a file F and then identify F with the next (encrypted) file uploaded by the client; (3) if the server can influence the queries of the client, or even if it knows some of the client’s queries, then the server can use that information to identify specific injected files with particular file identifiers. We postpone further discussion to Section 8.

In this paper, we focus only on query-recovery attacks where the server observes various tokens sent by the client followed the file identifiers returned, and the server’s goal is to determine the keywords corresponding to those tokens. This violates *query privacy*, which is important in its own right, and—as noted by Cash et al. [4]—can also be leveraged to violate *file privacy* since it reveals (some of) the keywords contained in (some of) the files. Our attacks show that the leakage of SE schemes should be analyzed carefully when SE is used as part of a larger system.

3 Binary-Search Attack

In this section, we present a basic query-recovery attack that we call the *binary-search attack*. This attack does not require the server to have any knowledge about the client’s files, and recovers all the keywords being searched by the client with 100% accuracy.

3.1 Basic Algorithm

The basic observation is that if the server injects a file F containing exactly half the keywords from the keyword universe K , then by observing whether the token t sent

by the client matches that file (i.e., whether F is returned in response to that token), the server learns one bit of information about the keyword corresponding to t . Using a standard non-adaptive version of binary search, the server can thus use $\lceil \log |K| \rceil$ injected files to determine the keyword exactly. The idea is illustrated in Figure 1 for $|K| = 8$.

The attack is described more formally in the pseudo-code of Figure 2. We assume for simplicity that $|K|$ is a power of 2, and identify K with the set $\{0, \dots, |K| - 1\}$ written in binary. The attack begins by having the server generate a set \mathbf{F} of $\log |K|$ files to be injected, where the i th file contains exactly those keywords whose i th most-significant bit is equal to 1. At some point,¹ the server learns, for each injected file, whether it is returned in response to some token t . We let $R = r_1 r_2 \dots$ denote the search results on the injected files, where $r_i = 1$ if and only if the i th file is returned in response to the token. For this attack, the server can deduce that the keyword corresponding to t is precisely R .

Algorithm $\mathbf{F} \leftarrow \text{Inject_Files}(K)$

- 1: **for** $i = 1, \dots, \log |K|$ **do**
- 2: Generate a file F_i that contains exactly the keywords in K whose i th bit is 1.
- 3: Output $\mathbf{F} = \{F_1, \dots, F_{\log |K|}\}$.

Algorithm $k \leftarrow \text{Recover}(R, K)$

- 1: Return R as the keyword from universe K associated with the token.

Figure 2: The binary-search attack. R denotes the search results for the token to be recovered on the injected files.

We highlight again that for this attack, the files are generated non-adaptively and independent of the token t . We note further that the *same* injected files can be used to recover the keywords corresponding to *any number of tokens*, i.e., once these files are injected, the server can recover the keywords corresponding to any future tokens sent by the client. The number of injected files needed for this attack is quite reasonable; with a 10,000-keyword universe, a server who sends only one email per day to the client can inject the necessary files in just 2 weeks.

Small keyword universe. For completeness and future reference, we note that the binary-search attack can be optimized if the hidden keyword is known to lie in some smaller universe of keywords, or if the server only cares about keywords lying in some subset of the entire keyword universe (and gives up on learning the keyword if

¹This can occur if the files are injected before the token t is sent, or if the files are injected after t is sent and the SE scheme does not satisfy forward privacy.

it lies outside this subset). Specifically, the server can carry out the binary-search attack from Figure 2 based on any subset $K' \subset K$ of the keyword universe using only $\log |K'|$ injected files.

3.2 Threshold Countermeasure

A prominent feature of the binary-search attack is that the files that need to be injected for the attack each contain a large number of keywords, i.e., $|K|/2$ keywords per file. We observe, then, that one possible countermeasure to our attack is to modify the SE scheme so as to limit the number of keywords per indexed file to some threshold $T \ll |K|/2$. This could be done either by simply not indexing files containing more than T keywords (possibly caching such files at the client), or by choosing at most T keywords to index from any file containing more than T keywords.

The threshold T can be set to some reasonably small value while not significantly impacting the utility of the SE scheme. For example, in the Enron email dataset [1] with roughly 5,000 keywords (see Section 5 for further details), the average number of keywords per email is 90; only 3% of the emails contain more than 200 keywords. Using the threshold countermeasure with $T = 200$ would thus affect only 3% of the honest client’s files, but would require the server to inject many more files in order to carry out a naive variant of the binary-search attack. Specifically, the server could replace each file F_i (that contains $|K|/2$ keywords) in the basic attack with a sequence of $|K|/2T$ files $F_{i,1}, \dots, F_{i,|K|/2T}$ each containing T keywords, such that $\cup_j F_{i,j} = F_i$. If any of these files is returned, this is equivalent to the original file F_i being returned in the basic attack. Note, however, that the server must now inject $|K|/2T \cdot \log |K|$ files. Unfortunately, as we explore in detail in the following section, the threshold countermeasure can be defeated using fewer injected files via more-sophisticated attacks.

Note also that the threshold countermeasure does not affect the binary-search attack with small keyword universe $K' \subset K$, as long as $|K'| \leq 2T$.

4 Advanced Attacks

In this section, we present more-sophisticated attacks for when the threshold countermeasure introduced in the previous section is used. In Section 4.1 we show an attack that uses fewer injected files than a naive modification of the binary-search attack, still without any knowledge of the client’s files. Then, in the following section, we show attacks that reduce the number of injected files even further, but based on the assumption that the server has information about some fraction of the client’s files.

4.1 Hierarchical-Search Attack

We noted earlier that the threshold countermeasure does not affect the binary-search attack with small keyword universe $K' \subset K$ if $|K'| \leq 2T$. We can leverage this to learn keywords in the *entire* universe using what we call a *hierarchical search attack*. This attack works by first partitioning the keyword universe into $\lceil |K|/T \rceil$ subsets containing T keywords each. The server injects files containing the keywords in each subset to learn which subset the client’s keyword lies in. In addition, it uses the small-universe, binary-search attack on adjacent pairs of these subsets to determine the keyword exactly. The algorithm is presented in Figure 3.

Algorithm $\mathbf{F} \leftarrow \text{Inject_Files_hierarchical}(K)$

- 1: Partition the universe into $w = \lceil |K|/T \rceil$ subsets K_1, \dots, K_w of T keywords each.
- 2: **for** $i = 1, 2, \dots, w$ **do**
- 3: Generate F_i containing every keyword $k \in K_i$.
- 4: **for** $i = 1, 2, \dots, w/2$ **do**
- 5: $\mathbf{F}_i \leftarrow \text{Inject_Files}(K_{2i-1} \cup K_{2i})$.
- 6: Output $\mathbf{F} = \{F_1, \dots, F_w, \mathbf{F}_1, \dots, \mathbf{F}_{w/2}\}$.

Algorithm $k \leftarrow \text{Recover_hierarchical}(R, K)$

- 1: Parse the search result R as

$$R = \{r_1, \dots, r_w, R_1, \dots, R_{w/2}\},$$

corresponding to the results on the files in \mathbf{F} described above.

- 2: Using the $\{r_i\}$, identify the subset $K_{2x-1} \cup K_{2x}$ the unknown keyword lies in.
- 3: $k \leftarrow \text{Recover}(R_x, K_{2x-1} \cup K_{2x})$.

Figure 3: The hierarchical-search attack. T is the threshold determining the maximum number of keywords in a file. R denotes the search results on the injected files. Inject_Files and Recover are from Figure 2.

We now calculate the number of injected files required by this attack. In Step 3 of $\text{Inject_Files_hierarchical}$, the server injects $\lceil |K|/T \rceil$ files, and in Step 5 it injects $\lceil |K|/2T \rceil \cdot \lceil \log 2T \rceil$ files. The total number of injected files is therefore at most

$$\lceil |K|/2T \rceil \cdot (\lceil \log 2T \rceil + 2).$$

In fact, for each i the first file in the set \mathbf{F}_i generated by $\text{Inject_files}(K_{2i-1} \cup K_{2i})$ is the same as F_{2i-1} and the server does not need to inject it again. Also, the server does not need to generate F_w in Step 3 because if the keyword is not in F_1, \dots, F_{w-1} then the server knows it must be in F_w . So the total number of injected files can

be improved to

$$\lceil |K|/2T \rceil \cdot (\lceil \log 2T \rceil + 1) - 1.$$

When the size of the keyword universe is $|K| = 5,000$ and the threshold is $T = 200$, the server needs to inject only 131 files, and the number of injected files grows linearly with the size of the keyword universe. We highlight again that the same injected files can be used to recover the keywords corresponding to any number of tokens; i.e., once these files are injected, the server can recover the keywords of any future searches made by the client.

We remark that an *adaptive* version of the above attack is also possible. Here, the attacker would first inject $\lceil |K|/T \rceil - 1$ files to learn what subset the unknown keyword lies in, and then carry out the small-universe, binary-search attack on a subset of size T . This requires only $\lceil |K|/T \rceil + \log T - 1$ injected files, but has the disadvantage of being adaptive and hence requires the SE scheme to not satisfy forward privacy. This version of the attack also has the disadvantage of targeting one particular search query of the client; additional files may need to be injected to learn the keyword used in some subsequent search query.

4.2 Attacks Using Partial Knowledge

With the goal of further decreasing the number of injected files required to recover a token in presence of the threshold countermeasure, we now explore additional attacks that leverage prior information that the server might have about some of the client’s files; we refer to the files known to the server as *leaked files*.² A similar assumption is used in prior work showing attacks on SE schemes [10, 4]; previous attacks, however, require the server to know about 90% of the client’s files to be effective (see Section 5), whereas our attacks work well even when the server knows a much smaller fraction of the client’s files.

Our attacks utilize the frequency of occurrence of the tokens and keywords in the client’s files. We define the *frequency* of a token (resp., keyword) as the fraction of the client’s files containing this token (resp., keyword). Similarly, we define the *joint frequency* of two tokens (resp., keywords) as the fraction of files containing *both* tokens (resp., keywords). The server learns the exact frequency (resp., joint frequency) of a token (resp., pair of tokens) based on the observed search results. The server obtains an estimate of the frequencies (resp., joint frequencies) of all the keywords based on the client’s files that it knows. We let $f(t)$ denote the exact (observed) frequency of token t , and let $f(t_1, t_2)$ be the joint frequency

²We stress that our attacks only rely on the *content* of these leaked files; we do not assume the server can identify the file identifiers corresponding to the leaked files after they have been uploaded to the server.

Algorithm $k \leftarrow \text{Inject_Files_Single}(t, K)$

- 1: Let K' be the set of $2T$ keywords with estimated frequencies closest to $f(t)$.
- 2: $\mathbf{F} \leftarrow \text{Inject_Files}(K')$.

Algorithm $k \leftarrow \text{Recover_Single}(R, K')$

- 1: If R contains all 0s, output \perp .
- 2: Else $k \leftarrow \text{Recover}(R, K')$.

Figure 4: Recovering a single keyword using partial file knowledge. T is the threshold determining the maximum number of keywords in a file. R denotes the search results on the injected files. `Inject_Files` and `Recover` are from Figure 2.

of tokens t_1, t_2 . We use $f^*(k)$ to denote the estimated frequency of keyword k , and define $f^*(k_1, k_2)$ analogously. Our attacks use the observation that if the leaked files are representative of all the client’s files, then $f(t)$ and $f^*(k)$ are close when t is the token corresponding to keyword k .

4.2.1 Recovering One Keyword

Say the server obtains a token t sent by the client, having observed frequency $f(t)$. The server first constructs a *candidate universe* K' for the keyword corresponding to t consisting of the $2T$ keywords whose estimated frequencies are closest to $f(t)$. The server then uses the small-universe, binary-search attack to recover the keyword exactly. In this way, the number of injected files is only $\lceil \log 2T \rceil$. The attack is presented in detail in Figure 4.

Differences from attacks in previous sections. The attack just described is *adaptive*, in that it targets a particular token t and injects files whose contents depend on the results of a search using t . This means the attack only applies to SE schemes that do not satisfy forward privacy. It also means that the attack needs to be carried out again in order to learn the keyword corresponding to some other token.

Another difference from our previous attacks is that this attack does not work with certainty. In particular, if the observed and estimated frequencies are far apart, or the number of keywords whose estimated frequencies are close to the observed frequency is larger than $2T$, the server may fail to recover the keyword corresponding to the token. On the other hand, the server can tell whether the attack succeeds or not, so will never associate an incorrect keyword with a token. This also means that if the attack fails, the attacker can re-run the attack with a different candidate universe, or switch to using one of our earlier attacks, in order to learn the correct keyword.

(We rely on this feature to design an attack for multiple tokens in the following section.) This is in contrast to earlier attacks [10, 4], where the attacker cannot always tell whether the keyword was recovered correctly.

4.2.2 Recovering Multiple Keywords

To learn the keywords corresponding to m tokens, the server can repeat the attack above for each token, but then the number of injected files will be (in the worst case) $m \cdot \lceil \log 2T \rceil$. A natural way to attempt to reduce the number of injected files is for the server to determine a candidate universe of size $2T$ for each token and then use the union of those candidate universes when injecting the files. In that case, however, the union would almost surely contain more than $2T$ keywords, in which case the number of keywords in the files produced by the binary-search attack will exceed the threshold T .

A second approach would be for the server to make the size of the candidate universe for each token $2T/m$, so the size of their union cannot exceed $2T$ keywords. Here, however, if m is large then the candidate universe for each token is very small and so the probability of the corresponding keyword not lying in its candidate universe increases substantially. Therefore, the recovery rate of this attack would be low.

Instead, we propose a more-complex attack that recovers multiple tokens by taking into account the joint frequencies for tokens and keywords. Our attack has two main steps (see Figure 5):

1. First, we recover the keywords corresponding to a subset of the tokens, namely the $n \ll m$ tokens with the highest observed frequencies. We recover the keywords using the second approach sketched above, which works (with few injected files) because n is small. This gives us as a set of tokens and their associated keywords as “ground truth.”
2. Given the ground truth, we recover the keyword associated with some other token t' using the following observation: if k' is the keyword corresponding to t' , then the observed joint frequency $f(t, t')$ should be “close” to the estimated joint frequency $f^*(k, k')$ for all pairs (t, k) in our ground-truth set, where “closeness” is determined by a parameter δ . By discarding candidate keywords that do not satisfy this property, we are left with a small set K' of candidate keywords for t' . If the candidate universe of keywords for each token is small enough, then even their union will be small. We then use a small-universe, binary-search attack to recover the corresponding keywords exactly.

Note that in the above attack the ability to tell whether a token is recovered correctly when building the ground

truth is crucial—otherwise the ground-truth set could contain many incorrect associations.

Parameter selection. Our attack has two parameters: n and δ . A larger value of n means that the ground-truth set can potentially be larger, but if n is too large then there is a risk that the candidate universe K_t (comprising the $2T/n$ keywords with estimated frequencies closest to $f(t)$) will not contain the true keyword corresponding to t . In our experiments, we set n heuristically to a value that achieves good performance.

The value of δ is chosen based on statistical-estimation theory. The estimated joint frequency is an empirical average computed from a collection of leaked files assumed to be sampled uniformly from the set of all files. Thus, we set δ such that if keywords k, k' correspond to tokens t, t' , respectively, then the estimated joint frequency $f^*(k, k')$ is within $\pm \delta \cdot f^*(k, k')$ of the true value $f(t, t')$ at least 99% of the time.

Ground-truth set selection. When building the ground-truth set, we recover the keywords associated with those tokens having the highest observed frequencies. We do so because those keywords can be recovered correctly with higher probability, as we explain next.

If the leaked files are chosen uniformly from the set of all files, then using statistical-estimation theory as above the attacker can compute a value δ such that at least 99% of the time it holds that $|f^*(k) - f(t)| \leq \varepsilon \cdot f^*(k)$, where k denotes the (unknown) keyword corresponding to t . Thus, if the attacker sets the candidate universe K_t to be the set of all keywords whose estimated frequencies are within distance $\varepsilon \cdot f^*(k)$ of $f(t)$, the candidate universe will include the keyword corresponding to t at least 99% of the time. The problem with taking this approach is that the set K_t constructed this way may be too large.

If we assume a Zipfian distribution [3] for the keyword frequencies, however, then the size of K_t as constructed above is smallest when $f(t)$ is largest. (This is a consequence of the fact that the Zipfian distribution places high probability on a few items and low probability on many items.) In particular, then, the set of $2T/n$ keywords with estimated frequencies closest to $f(t)$ (as chosen by our algorithm), will “cover” all keywords within distance $\varepsilon \cdot f^*(k)$ of $f(t)$ from $f(t)$ – or, equivalently, the candidate universe will contain the true keyword k – with high probability.

5 Experiments

We simulate the attacks from Section 4. (We do not run any simulations for the binary-search attack described in Section 3, since this attack succeeds with probability 1, injecting a fixed number of emails.) We compare our attacks to our own implementation of the attacks by Cash

$\mathbf{t} = \{t_1, \dots, t_m\}$ is the set of m tokens whose keywords we wish to recover.

Algorithm $\mathbf{k} \leftarrow \text{Attack_Multiple_Tokens}(\mathbf{t}, K)$

Build ground truth set G .

- 1: Sort tokens in \mathbf{t} according to their exact frequencies $f(t)$. Let \mathbf{t}_1 denote the n tokens with highest observed frequencies.
- 2: **for** each token t in \mathbf{t}_1 **do**
- 3: Set its candidate universe K_t as the set of $\frac{2T}{n}$ keywords with estimated frequencies $f^*(k)$ nearest to $f(t)$.
- 4: Define $K' = \cup_{t \in \mathbf{t}_1} K_t$ and inject files generated by $\mathbf{F}_1 \leftarrow \text{Inject_Files}(K')$.
- 5: **for** each token t in \mathbf{t}_1 **do**
- 6: Let R_t be the search result of token t on files \mathbf{F}_1 .
- 7: **if** R_t is not all 0s **then**
- 8: $k_t \leftarrow \text{Recover}(R_t, K')$.
- 9: Add (t, k_t) to G .

Recover the remaining tokens, let \mathbf{t}_2 be the set of unrecovered tokens.

- 10: **for** each token $t' \in \mathbf{t}_2$ **do**
- 11: Set its candidate universe $K_{t'}$ as the set of $2T$ keywords with estimated frequencies $f^*(k)$ nearest to $f(t')$.
- 12: **for** each keyword $k' \in K_{t'}$ **do**
- 13: **for** each token/keyword pair $(t, k) \in G$ **do**
- 14: If $|f(t, t') - f^*(k, k')| > \delta \cdot f^*(k, k')$, remove k' from candidate universe $K_{t'}$.
- 15: Set $K'' = \cup_{t' \in \mathbf{t}_2} K_{t'}$.
- 16: **if** $|K''| \leq 2T$ **then**
- 17: $\mathbf{F}_2 \leftarrow \text{Inject_Files}(K'')$.
- 18: **for** each token $t' \in \mathbf{t}_2$ **do**
- 19: Let $R_{t'}$ be the search result of token t' on files \mathbf{F}_2 .
- 20: $k_{t'} \leftarrow \text{Recover}(R_{t'}, K'')$
- 21: **else**
- 22: $\mathbf{F}_2 \leftarrow \text{Inject_Files_hierarchical}(K'')$.
- 23: **for** each token $t' \in \mathbf{t}_2$ **do**
- 24: Let $R_{t'}$ be the search result of token t' on files \mathbf{F}_2 .
- 25: $k_{t'} \leftarrow \text{Recover_hierarchical}(R_{t'}, K'')$
- 26: Output \mathbf{k} that includes all recovered keywords.

Figure 5: Recovering multiple keywords using partial file knowledge. T is the threshold determining the maximum number of keywords in a file; δ is a parameter. Inject_Files and Recover are from Figure 2.

et al. [4] (CGPR15). We do not compare with the attacks of Islam et al. [10] (IKK12), since their results are strictly dominated by those of CGPR15.

5.1 Setup

For our experiments we use the Enron email dataset [1], consisting of 30,109 emails from the “sent mail” folder of 150 employees of the Enron corporation that were sent between 2000–2002. We extracted keywords from this dataset as in CGPR15: words were first stemmed using the standard Porter stemming algorithm [18], and we then removed 200 stop words such as “to,” “a,” etc. Doing so results in approximately 77,000 keywords in total. In our experiments, we chose the top 5,000 most frequent

keywords as our keyword universe (as in CGPR15).

We assumed the threshold countermeasure with $T = 200$. As discussed earlier, only 3% of the files contained more than this many keywords.

We could not find real-world query datasets for email. Therefore, in our experiments we choose the client’s queries uniformly from the keyword universe, as in CGPR15. (However, our attacks do not use any information about the distribution of the queries.) Leaked files are chosen uniformly from the base set of 30,109 emails, and the percentage of leaked files was varied from 1% to 100%. For each value of the file-leakage percentage, we repeat the attack on 100 uniform sets of queries (containing either one token or 100 tokens) and 10 uniformly sampled sets of leaked files of the appropriate size; we

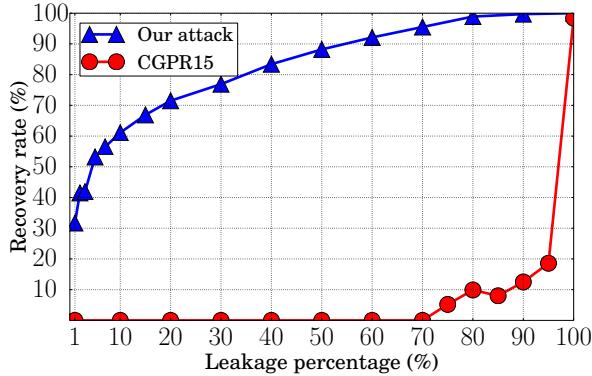


Figure 6: Recovering the keyword corresponding to a single token. Probability of recovering the correct keyword as a function of the percentage of files leaked.

report the average. We do not include error bars in our figures, but have observed that the standard deviation in our experiments is very small (less than 3% of the average).

5.2 Recovery of a Single Token

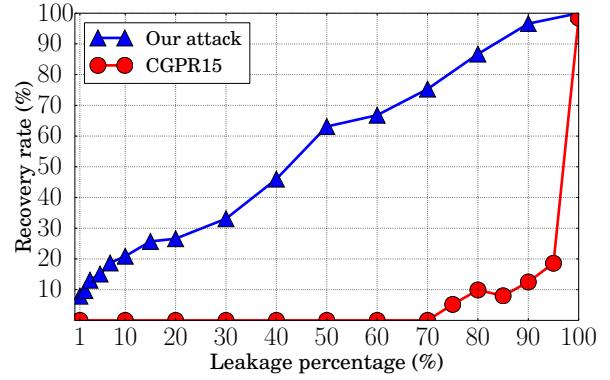
The performance of our attack for recovering the keyword associated with a single token (described in Section 4.2.1) is displayed in Figure 6. The server only needs to inject $\lceil \log 2T \rceil = 9$ files in order to carry out the attack. It can be observed that our attack performs quite well even with only a small fraction of leaked files, e.g., recovering the keyword about 70% of the time once only 20% of the files are leaked, and achieving 30% recovery rate even when given only 1% of the files.

Neither the IKK12 attack nor the CGPR15 attack applies when the server is given the search results of only a single token. To provide a comparison with our results, we run the CGPR15 attack by giving it the search results of 100 tokens (corresponding to uniformly chosen keywords) and then measure the fraction of keywords recovered. As shown in Figure 6, the CGPR15 attack recovers a keyword with probability less than 20% even when 95% of the client’s files are leaked. Of course, our attack model is stronger than the one considered in CGPR15.

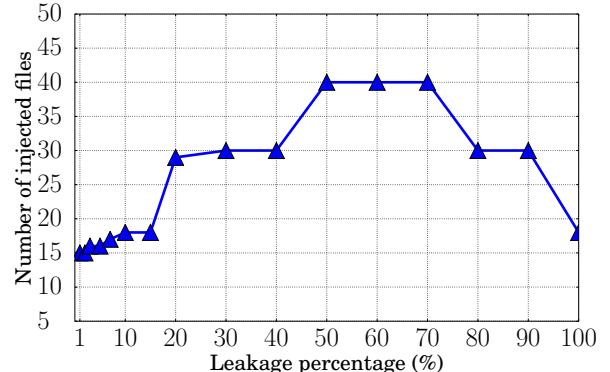
5.3 Recovery of Multiple Tokens

We have also implemented our attack from Section 4.2.2 which can be used to recover the keywords corresponding to multiple tokens. In our experiments, we target the recovery of the keywords associated with $m = 100$ tokens; we choose $n = 10$, and set δ as described in Section 4.2.2.

Figure 7a tabulates the fraction of keywords recovered by our attack, and compares it to the fraction recovered



(a)



(b)

Figure 7: Recovering the keywords corresponding to 100 tokens. (a) Fraction of keywords recovered and (b) number of files injected as a function of the percentage of files leaked.

by the CGPR15 attack. (As noted in the previous section, the CGPR15 attack inherently requires search results for multiple tokens; this explains why the results for the CGPR15 attack in Figure 7a are almost identical to the results for their attack in Figure 6.) Both attacks do well when the fraction of leaked files is large, however the recovery rate of the CGPR15 attack drops dramatically as the fraction of leaked files decreases. In contrast, our attack continues to perform well, recovering 65% of the keywords given access to 50% of the client’s files, and still recovering 20% of the keywords when only 10% of the client’s files have been leaked. We stress that in our attack the server knows which keywords have been recovered correctly and which have not, something that is not the case for prior attacks.

Figure 7b shows the number of files that need to be injected in order to carry out our attack. The number of files injected never exceeds 40, and in many cases it is even less than that. We also highlight that the number of files injected to recover the keywords associated with 100

tokens is more than an order-of-magnitude smaller than $100 \times$ the number of files injected to recover the keyword associated with a single token in the previous section.

The number of files injected by our attack first increases with the fraction of leaked files, and then decreases; we briefly explain why. The number of files injected in step 1 of our attack is independent of the fraction of leaked files. The number of files injected in step 2 of the attack depends on both the number of unrecovered tokens (i.e., the size of \mathbf{t}_2) and the average size of the candidate universe for each unrecovered token t' (i.e., the size of $K_{t'}$). When the fraction of leaked files is very small, the estimated joint frequencies are far from the true frequencies and, in particular, most estimated joint frequencies are 0; thus, many keywords are removed from $K_{t'}$ and hence the size of $K_{t'}$ is low. The net result is that the recovery rate is small, but so is the number of injected files. As the fraction of leaked files increases, more keywords are included in $K_{t'}$, leading to higher recovery rate but also more injected files. When the fraction of leaked files becomes very high, however, the estimated frequencies are very close to the true frequencies and so more keywords are recovered in step 1 of the attack. This leaves fewer unrecovered tokens in step 2, leading to fewer injected files overall even as the recovery rate remains high.

6 Ineffectiveness of Keyword Padding

Prior work [10, 4] suggests *keyword padding* as another potential countermeasure for attacks that exploit the file-access pattern. The basic idea is to distort the real frequency of each keyword k by randomly associating files that do not contain that keyword with k ; this is done at setup time, when the client uploads its encrypted files to the server. One version of the countermeasure [4] ensures that the number of files returned in response to any search result is a multiple of an integer λ . A stronger version of the countermeasure [10] involves performing the padding in such a way that for any keyword k there are at least $\alpha - 1$ other keywords having the same frequency. These countermeasures defeat the attacks in prior work, but we show that they have little effect on our attacks.

We remark that keyword padding seems difficult to apply in the dynamic setting, where new files are uploaded after the initial setup done by the client. The dynamic case is not discussed in [10, 4].

6.1 Binary-/Hierarchical-Search Attacks

Even when keyword padding is used, our binary-search and hierarchical-search attacks will recover the keyword k corresponding to some token t unless one of the

injected files that does not contain k is returned in response to the search using t . We show that the probability of this bad event is small, focusing on the binary-search attack for concreteness. Say ℓ of the files contain k and that, after keyword padding, an additional $\beta \cdot \ell$ random and independently chosen files (in expectation) that do not contain k are returned in response to the search using t . (By setting parameters appropriately, this roughly encompasses both the countermeasures described above.) Now consider some file injected as part of the binary-search attack that does not contain k . The probability that this file is chosen as one of the spurious files returned in response to the search using t is $\beta\ell/(F - \ell)$, where F is the total number of files (including the injected files). Since $\lceil \log |K| \rceil$ files are injected, the overall probability that the bad event occurs is at most

$$1 - \left(1 - \frac{\beta\ell}{(F - \ell)}\right)^{\lceil \log |K| \rceil}.$$

In fact, this is an over-estimate since if k is uniform then on average only half the injected files contain k .

For the Enron dataset with $|K| = 5,000$, $F = 30,109$, $\ell = 560$, and $\beta = 0.6$, and assuming half the injected files contain the keyword in question, the probability that the binary-search attack succeeds is 0.93. (In fact, $\beta = 0.6$ is quite high, as this means that more than 1/3 of the files returned in response to a query do not actually contain the searched keyword.) With $\beta = 0.6$ the IKK12 and CGPR15 attacks recover no keywords at all.

6.2 Attacks with Partial File Leakage

Although our attacks with partial file leakage use information about keyword frequencies and joint frequencies, they are still not significantly affected by the padding countermeasures. The reason is that although the padding ensures that a given frequency no longer suffices to uniquely identify a keyword, the frequency of any particular keyword doesn't change very much. Thus, the exact frequency and the estimated frequency of any keyword remain close even after the padding is done, and the underlying keyword is still likely to be included in the candidate universe of a target token. As long as this occurs, the search step recovers the token with high probability as discussed in the previous section. This is even more so the case with regard to joint frequencies, since these do not change unless two keywords are both associated with the same random file that contains neither of those keywords, something that happens with low probability.

To validate our argument, we implement the padding countermeasure proposed in [4] and repeat the experiments using our attacks. As shown in Figures 8 and 9,

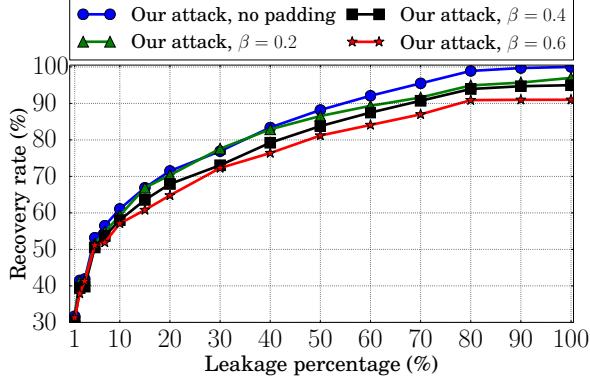


Figure 8: Recovering the keyword corresponding to a single token when keyword padding is used.

the recovery rate of our attacks degrades only slightly when keyword padding is used.

Figure 10 compares the effectiveness of our attack to the CGPR15 attack when keyword padding is used. The recovery rate of the CGPR15 attack drops dramatically in the presence of this countermeasure. In particular, it recovers only 57% of the tokens even with 100% file leakage when $\beta = 0.2$, and recovers nothing even with 100% file leakage when $\beta = 0.6$. In contrast, our attack still recovers almost the same number of keywords as when no padding is used.

7 Extensions to Conjunctive SE

SE schemes supporting conjunctive queries allow the client to request all files containing some collection of keywords k_1, k_2, \dots, k_d . The naive way to support conjunctive queries is to simply have the client issue queries for each of these keywords individually; the server can compute the set of file identifiers S_i containing each keyword k_i and then take their intersection to give the final result. Such an approach leaks more information than necessary: specifically, it leaks each of S_1, \dots, S_d rather than the final result $\cap S_i$ alone. We refer to $\cap S_i$ as the *ideal access-pattern leakage* for a conjunctive query, and show attacks based only on such ideal leakage. We remark, however, that no known efficient SE scheme achieves ideal leakage. For example, the scheme by Cash et al. [6] leaks $S_1, S_1 \cap S_2, S_1 \cap S_3, \dots, S_1 \cap S_d$. Such additional leakage can only benefit our attacks.

Throughout this section, we assume the threshold countermeasure is not used and so injected files can contain any number of keywords. (Our attacks here could be generalized as done previously in case the threshold countermeasure is used.)

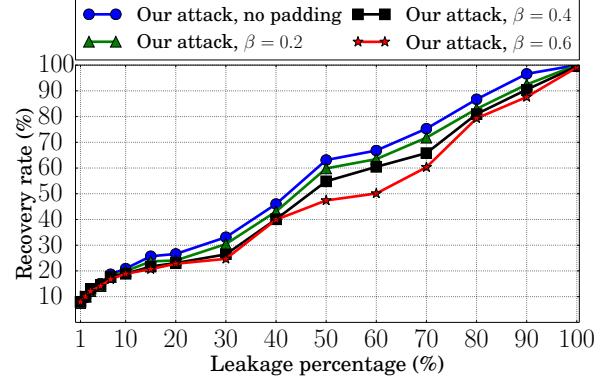


Figure 9: Recovering the keywords corresponding to 100 tokens when keyword padding is used.

7.1 Queries with Two Keywords

We first present a non-adaptive attack to recover the keywords used in a conjunctive query involving two keywords. As in the non-adaptive attacks in prior sections, the attacker can recover the keywords corresponding to any future queries after injecting some initial set of files.

The idea is the following. Say the conjunctive search query involves keywords k_1 and k_2 , and we can partition the universe of keywords into two sets K_1 and K_2 with $k_1 \in K_1$ and $k_2 \in K_2$. We can then use a variant of the binary-search attack in which we inject files generated by `Inject_Files(K_1)`, where we additionally include all keywords in K_2 . Since these files always contain k_2 , the search results of the conjunctive query on these injected files is exactly the same as the search results of k_1 on these files, and we can thus recover k_1 as before. We can proceed analogously to recover k_2 .

The problem with the above is that we do not know, *a priori*, how to partition K into sets K_1, K_2 as required. Instead, we generate a sequence of $\log |K|$ partitions $\{(K_1^i, K_2^i)\}$ such that for some partition i it holds that $k_1 \in K_1^i$ and $k_2 \in K_2^i$. This is done by simply letting K_1^i be the set of all keywords whose i th bit is 0, and K_2^i be the complement. Since k_1 and k_2 are distinct, they must differ on at least one position, say i , and satisfy the desired separation property on the i th partition. By repeating the attack described earlier for each partition, we obtain an attack using $\log^2 |K| + \log |K|$ injected files (after removing duplicates). The attack is described in detail in Figure 11.

Given ideal access-pattern leakage, the above attack above only works for conjunctive queries involving two keywords. For conjunctive searches using the SE scheme of Cash et al. [6], though, the above attack can be extended to work for conjunctive queries involving any number of keywords since the pairwise intersections are leaked as described earlier.

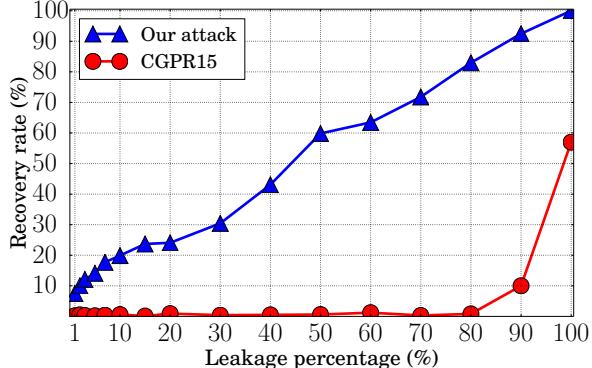
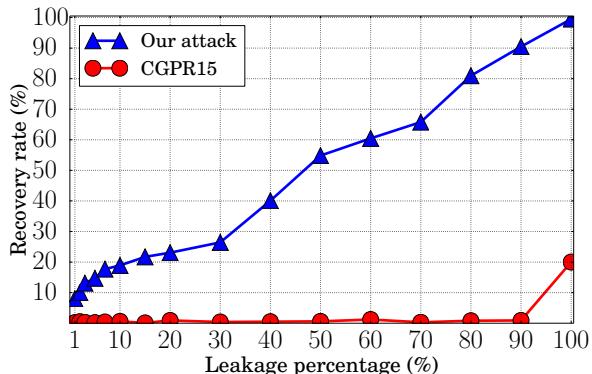
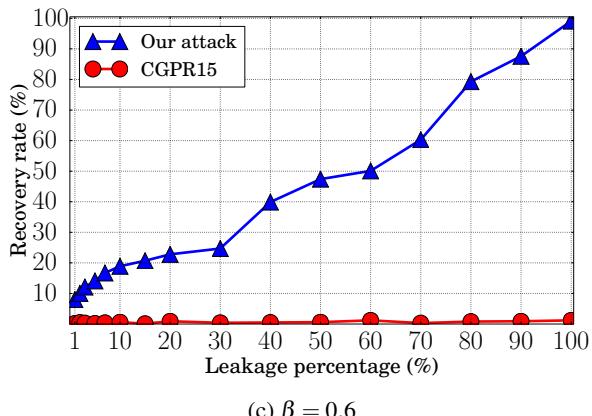
(a) $\beta = 0.2$ (b) $\beta = 0.4$ 

Figure 10: Recovering the keywords corresponding to 100 tokens when keyword padding is used, plotted for different β .

7.2 Queries with Multiple Keywords

The attack in Section 7.1 only works for conjunctive queries involving two keywords, and uses $O(\log^2 |K|)$ injected files. Here we present a non-adaptive attack that can recover conjunctive queries involving any number of keywords using only $O(\log |K|)$ injected files, and still assuming only ideal access-pattern leakage. In contrast

Let q be a conjunctive query with two keywords.

Algorithm F \leftarrow Inject_Files_Disjoint(K_1, K_2)

- 1: $\mathbf{F} \leftarrow$ Inject_Files(K_1).
- 2: Include all keywords in K_2 in every file in \mathbf{F} .

Algorithm F \leftarrow Inject_Files_Conjunctive(K)

- 1: **for** $i = 1, 2, \dots, \log |K|$ **do**
- 2: Let K_1^i contain keywords whose i th bit is 0, and let $K_2^i = K \setminus K_1^i$.
- 3: Generate file F_1^i that contains all keywords in K_1^i and file F_2^i that contains all keywords in K_2^i .
- 4: $\mathbf{F}_1^i \leftarrow$ Inject_Files_Disjoint(K_1^i, K_2^i).
- 5: $\mathbf{F}_2^i \leftarrow$ Inject_Files_Disjoint(K_2^i, K_1^i).
- 6: Output $\mathbf{F} = \{F_1^i, F_2^i, \mathbf{F}_1^i, \mathbf{F}_2^i\}$, for all i .

Algorithm k \leftarrow Recover_Conjunctive(q, K, \mathbf{F})

- 1: Let $R_q = \{r_1^i, r_2^i, R_1^i, R_2^i\}$ for $i = 1, \dots, \log |K|$ be the search result of query q on the files \mathbf{F} described above.
- 2: Find i such that neither F_1^i nor F_2^i is in the search result (i.e., $r_1^i = r_2^i = 0$).
- 3: $k_1 \leftarrow$ Recover(R_1^i, K_1^i).
- 4: $k_2 \leftarrow$ Recover(R_2^i, K_2^i).
- 5: Output (k_1, k_2) .

Figure 11: Non-adaptive attack for a conjunctive query involving two keywords.

to the previous attack, however, this attack does not always succeed.

Consider a conjunctive query q involving d keywords. The basic idea is to inject n files, each containing L keywords selected uniformly and independently from the keyword universe. If parameters are set appropriately, the search result on q will include some of the injected files with high probability. By definition, each of those files contains all d keywords involved in the query, and hence the intersection of those files also contains all those keywords. We claim that when parameters are set appropriately, the intersection contains no additional keywords. Thus, the server recovers precisely the d keywords involved in the query by simply taking the intersection of the injected files returned in response to the query. The following theorem formalizes this idea.

Theorem 1. Let $L = (\frac{1}{2})^{1/d}|K|$ and $n = (2 + \varepsilon)d \log |K|$ with $\varepsilon \geq 0$. Then the success probability of the attack is roughly $e^{-1/|K|^{\varepsilon/4}}$.

Proof. Fix some conjunctive query q involving d keywords. The probability that any particular injected file matches the query is approximately $(L/|K|)^d = 1/2$ since each of the d keywords is included in the file with

probability roughly $L/|K|$. Since each file is generated independently, the expected number of files that match the query is $n/2$; moreover, the number n' of files that match the query follows a binomial distribution and so the Chernoff bound implies

$$\Pr \left[\left| n' - \frac{n}{2} \right| \geq \theta \frac{\sqrt{n}}{2} \right] \leq e^{-\theta^2/2}.$$

Setting $\theta = \frac{\varepsilon\sqrt{n}}{2(2+\varepsilon)}$, we have

$$\Pr \left[n' \leq \left(1 + \frac{\varepsilon}{4} \right) d \log |K| \right] \leq e^{-\frac{\theta^2}{2}}.$$

Thus, $n' > (1 + \frac{\varepsilon}{4})d \log |K|$ with overwhelming probability.

The probability that any *other* keyword is in all these n' files is extremely low. Specifically, for any fixed keyword not involved in the query, the probability that it lies in all n' files is $(L/|K|)^{n'}$. Thus, the probability that *no* other keyword lies in all n' files is

$$\left(1 - \left(\frac{L}{|K|} \right)^{n'} \right)^{|K|-d} \approx \left(1 - \frac{1}{|K|^{1+\varepsilon/4}} \right)^{|K|}$$

(assuming $d \ll |K|$). The above simplifies to $e^{-1/|K|^{\varepsilon/4}}$. \square

Note that for any $\varepsilon > 0$ the bound given by the theorem approaches 1 as $|K|$ tends to infinity. We experimentally verified the bound in the theorem for $|K| = 5,000$ and $d = 3$. For example, setting $\varepsilon = 1$ we obtain an attack in which the server injects $n = 110$ files with $L = 3,969$ keywords each, and recovers all keywords involved in the conjunctive query with probability 0.97. For completeness, we remark that the server can tell whether it correctly recovers all the keywords or not, assuming d is known.

7.3 An Adaptive Attack

We can further reduce the number of injected files using an adaptive attack. The idea is to recover the keywords involved in the query one-by-one, starting with the lexicographically largest, using an adaptive binary search for each keyword. The server first injects a file containing the first $|K|/2$ keywords. There are two possibilities:

1. If this file is in the search result for the query, the server learns that all the keywords involved in the query have index at most $|K|/2$. It will next inject a file containing the first $|K|/4$ keywords.
2. If this file is not in the search result for the query, the server learns that at least one keyword involved in the query has index greater than $|K|/2$. It will next inject a file containing the first $3|K|/4$ keywords.

Proceeding in this way, the server learns the lexicographically largest keyword using $\log |K|$ injected files. Once that keyword k_d is recovered, the server repeats this attack but with k_d always included in the injected files to learn the next keyword, and so on. See Figure 12.

Let q be a conjunctive query with keyword k_1, \dots, k_d .

Algorithm $\mathbf{k} \leftarrow \text{Attack_Conjunctive}(q, K)$

```

1: Initialize  $\mathbf{k} = \emptyset$ .
2: for  $i = d, \dots, 1$  do
3:   Set  $K_i = K \setminus \mathbf{k}$ , set  $b = |K_i|/2$ .
4:   for  $j = 2, \dots, \log |K_i|$  do
5:     Inject  $F$  that contains the first  $b$  keywords
       in  $K_i$  and all keywords in  $\mathbf{k}$ .
6:     Let  $R_q$  be the search result of query  $q$  on  $F$ .
7:     if  $R_q = 1$  then
8:        $b = b - |K_i|/2^j$ .
9:     else
10:       $b = b + |K_i|/2^j$ .
11: Inject  $F$  that contains the first  $b$  keywords in  $K_i$ 
       and all keywords in  $\mathbf{k}$ .
12: Let  $R_q$  be the search result of query  $q$  on  $F$ .
13: if  $R_q = 1$  then
14:   Recover  $k_i$  as the  $b$ th keyword in  $K_i$ .
15: else
16:   Recover  $k_i$  as the  $(b+1)$ th keyword in  $K_i$ .
17:  $\mathbf{k} = \mathbf{k} \cup \{k_i\}$ .

```

Figure 12: An adaptive attack for conjunctive queries involving d keywords.

The number of injected files is $d \log |K|$. We remark that d need not be known in advance, since the attacker can determine d during the course of the attack. It is also worth observing that the number of injected files is essentially optimal for a deterministic attack with success probability 1, because the search results on $d \log |K|$ files contain at most $d \log |K|$ bits of information, which is roughly the entropy of a conjunctive search involving d keywords from a universe of size $|K|$.

8 Additional (Potential) Countermeasures

In this section, we briefly discuss some other potential countermeasures against our attacks.

Semantic filtering. One may be tempted to think that the files injected by our attacks will not “look like” normal English text, and can therefore be filtered easily by the client. We argue that such an approach is unlikely to prevent our attacks. First, although as described our attacks inject files containing arbitrary sets of keywords, the

server actually has some flexibility in the choice of keywords; e.g., the binary-search attack could be modified to group sets of keywords that appear naturally together. Second, within each injected file, the server can decide the order and number of occurrences of the keywords, can choose variants of the keywords (adding “-ed” or “-s,” for example), and can freely include non-keywords (“a,” “the,” etc.) There are several tools (e.g., [21]) that can potentially be adapted to generate grammatically correct text from a given set of keywords by ordering keywords based on *n*-grams trained from leaked files and simple grammatical rules. A detailed exploration is beyond the scope of our paper.

Batching updates. As mentioned in Section 2, even if the client shuffles the file identifiers and pads all files to the same length, the server can identify an injected file based on the time at which it is inserted by the client. This suggests a (partial) countermeasure that can be used in dynamic SE schemes that support updates: rather than uploading each new file as it arrives, the client should wait until there are several (say, B) new files and then upload this “batch” of B files at once. Assuming only one of those files was injected by the server, this means the server only learns that the injected file corresponds to one of B possibilities.

This countermeasure can be trivially circumvented if the server can inject B files before any other new files arrive. (If the server additionally has the ability to mount chosen-query attacks—something we have not otherwise considered in this paper—then the total number of injected files remains the same.) Even if the server can inject only $B' < B$ identical³ files into a single “batch,” the server knows that if fewer than B' files from this batch are returned in response to some query, then the injected files do not match that query. Finally, even if the server can only inject a single file per “batch,” the server can inject the same file repeatedly and with high confidence determine based on the search results whether the file matches some query. We leave a more complete analysis of this countermeasure for future work.

9 Conclusions

Our paper shows that file-injection attacks are devastating for query privacy in searchable encryption schemes that leak file-access patterns. This calls into question the utility of searchable encryption, and raises doubts as to whether existing SE schemes represent a satisfactory tradeoff between their efficiency and the leakage they allow. Nevertheless, we briefly argue that searchable encryption may still be useful in scenarios where

³The files need not be identical; they only need to contain an identical set of keywords.

file-injection attacks are not a concern, and then suggest directions for future research.

We have argued that file-injection attacks would be easy to carry out in the context of searching email. But in “closed systems,” where a client is searching over records generated via some other process, file-injection attacks may not be possible or may be much more difficult to carry out. Additionally, there may be settings—e.g., when all files have the same length because they share some particular format—where even though the server can inject files, it may not be able to associate file identifiers with specific files it has injected. It is worth noting also that there may be applications of SE in which the server is trusted (and so, in particular, can be assumed not to carry out file-injection attacks), and the threat being defended against is an external attacker who compromises the server.⁴

Our work and previous work [10, 4] demonstrate that leaking file-access patterns in their entirety is dangerous, and can be exploited by an attacker to learn a significant amount of sensitive information. We suggest, therefore, that future research on searchable encryption focus on reducing or eliminating this leakage rather than accepting it as the default. Our work also highlights the need to design efficient schemes satisfying forward privacy. Addressing these challenges may require exploring new directions, such as interactive protocols [17] or multiple servers. It would also be of interest to explore lower bounds on the efficiency that searchable encryption can achieve as a function of how much about the file-access pattern is leaked.

References

- [1] Enron email dataset. <https://www.cs.cmu.edu/~./enron/>. Accessed: 2015-12-14.
- [2] Pmail. <https://github.com/tonypr/Pmail>, 2014.
- [3] ADAMIC, L. A., AND HUBERMAN, B. A. Zipfs law and the internet. *Glottometrics* 3, 1 (2002), 143–150.
- [4] CASH, D., GRUBBS, P., PERRY, J., AND RISTENPART, T. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 668–679.
- [5] CASH, D., JAEGER, J., JARECKI, S., JUTLA, C. S., KRAWCZYK, H., ROSU, M.-C., AND STEINER, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. *IACR Cryptology ePrint Archive* 2014 (2014), 853.
- [6] CASH, D., JARECKI, S., JUTLA, C., KRAWCZYK, H., ROŞU, M.-C., AND STEINER, M. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology—CRYPTO 2013*. Springer, 2013, pp. 353–373. Full version available at <http://eprint.iacr.org>.

⁴Though even here one must be careful since an external attacker might have the ability to inject files, and/or be able to learn file-access patterns from the client-server communication (e.g., based on file lengths) without compromising the server.

- [7] CHANG, Y.-C., AND MITZENMACHER, M. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security* (2005), Springer, pp. 442–455.
- [8] CURTMOLA, R., GARAY, J., KAMARA, S., AND OSTROVSKY, R. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security* (2006), ACM, pp. 79–88.
- [9] GOH, E.-J., ET AL. Secure indexes. *IACR Cryptology ePrint Archive* 2003 (2003), 216.
- [10] ISLAM, M. S., KUZU, M., AND KANTARCIOLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012* (2012).
- [11] KAMARA, S., AND PAPAMANTHOU, C. Parallel and dynamic searchable symmetric encryption. In *Financial cryptography and data security*. Springer, 2013, pp. 258–274.
- [12] KAMARA, S., PAPAMANTHOU, C., AND ROEDER, T. Dynamic searchable symmetric encryption. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012* (2012), pp. 965–976.
- [13] LAU, B., CHUNG, S., SONG, C., JANG, Y., LEE, W., AND BOLDYREVA, A. Mimesis aegis: A mimicry privacy shield—a systems approach to data privacy on public cloud. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 33–48.
- [14] LIU, C., ZHU, L., WANG, M., AND TAN, Y.-A. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences* 265 (2014), 176–188.
- [15] NAVEED, M. The fallacy of composition of oblivious ram and searchable encryption. Tech. rep., Cryptology ePrint Archive, Report 2015/668, 2015.
- [16] NAVEED, M., PRABHAKARAN, M., AND GUNTER, C. A. Dynamic searchable encryption via blind storage. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 639–654.
- [17] POPA, R. A., LI, F. H., AND ZELDOVICH, N. An ideal-security protocol for order-preserving encoding. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 463–477.
- [18] PORTER, M. F. An algorithm for suffix stripping. *Program* 14, 3 (1980), 130–137.
- [19] SONG, D. X., WAGNER, D., AND PERRIG, A. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on* (2000), IEEE, pp. 44–55.
- [20] STEFANOV, E., PAPAMANTHOU, C., AND SHI, E. Practical dynamic searchable encryption with small leakage. In *NDSS* (2014), vol. 14, pp. 23–26.
- [21] UCHIMOTO, K., ISAHARA, H., AND SEKINE, S. Text generation from keywords. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1* (2002), Association for Computational Linguistics, pp. 1–7.

Investigating Commercial Pay-Per-Install and the Distribution of Unwanted Software

Kurt Thomas[◊] Juan A. Elices Crespo[◊] Ryan Rasti[◊] Jean-Michel Picod[◊] Cait Phillips[◊]
Marc-André Decoste[◊] Chris Sharp[◊] Fabio Tirelo[◊] Ali Tofigh[◊] Marc-Antoine Courteau[◊]
Lucas Ballard[◊] Robert Shield[◊] Nav Jagpal[◊] Moheeb Abu Rajab[◊] Panayiotis Mavrommatis[◊]
Niels Provos[◊] Elie Bursztein[◊] Damon McCoy^{†*}

[◊]*Google* [†]*New York University* ^{*}*International Computer Science Institute*

Abstract

In this work, we explore the ecosystem of *commercial pay-per-install* (PPI) and the role it plays in the proliferation of unwanted software. Commercial PPI enables companies to bundle their applications with more popular software in return for a fee, effectively commoditizing access to user devices. We develop an analysis pipeline to track the business relationships underpinning four of the largest commercial PPI networks and classify the software families bundled. In turn, we measure their impact on end users and enumerate the distribution techniques involved. We find that unwanted ad injectors, browser settings hijackers, and “cleanup” utilities dominate the software families buying installs. Developers of these families pay \$0.10–\$1.50 per install—upfront costs that they recuperate by monetizing users without their consent or by charging exorbitant subscription fees. Based on Google Safe Browsing telemetry, we estimate that PPI networks drive over 60 million download attempts every week—nearly three times that of malware. While anti-virus and browsers have rolled out defenses to protect users from unwanted software, we find evidence that PPI networks actively interfere with or evade detection. Our results illustrate the deceptive practices of some commercial PPI operators that persist today.

1 Introduction

In recent years, *unwanted software* has risen to the forefront of threats facing users. Prominent strains include ad injectors that laden a victim’s browser with advertisements, browser settings hijackers that sell search traffic, and user trackers that silently monitor a victim’s browsing behavior. Estimates of the incident rate of unwanted software installs on desktop systems are just emerging: prior studies suggest that ad injection affects as many as 5% of browsers [34] and that deceptive extensions escaping detection in the Chrome Web Store affect over 50 million users [17].

Despite the proliferation of unwanted software, the root source of installs remains unclear. One potential explanation is *commercial pay-per-install* (PPI), a monetization scheme where software developers bundle several third-party applications as part of their installation process in return for a payout. We differentiate this from *blackmarket pay-per-install* [4] as commercial PPI relies on a user consent dialogue to operate aboveboard. Download portals are a canonical example, where carelessly installing any of the top applications may leave a system bloated with search toolbars, anti-virus free trials, and registry cleaners [16]. Unfortunately, this all too common user experience is the profit vehicle for a collection of private and publicly companies that commoditize software bundling [15]. While earnings in this space are nebulous, one of the largest commercial PPI outfits reported \$460 million in revenue in 2014 [31].

In this work, we explore the ecosystem of commercial PPI and the role it plays in distributing the most notorious unwanted software families. The businesses profiting from PPI operate *affiliate networks* to streamline buying and selling installs. We identify a total of 15 PPI affiliate networks headquartered in Israel, Russia, and the United States. We select four of the largest to investigate, monitoring each over a year long period from January 8, 2015–January 7, 2016 in order to track the software families paying for installs, their impact on end users, and the deceptive distribution practices involved.

We find that commercial PPI distributes roughly 160 software families each week, 59% of which at least one anti-virus engine on VirusTotal [36] flags as unwanted. For our study, we use this labeling to classify unwanted software. The families with the longest PPI distribution campaigns include ad injectors, like Crossrider, and scareware that dupes victims into paying a subscription fee for resolving “dangerous” registry settings, a hair’s length shy of ransomware. We find that PPI networks support unwanted software as first-class partners: down-

loaders will actively fingerprint a victim’s machine in order to detect hostile anti-virus or virtualized environments, in turn dynamically selecting offers that go undetected. Software developers pay between \$0.10–1.50 per install for these services, where price is dictated by geographic demand.

Via Safe Browsing telemetry, we measure the impact of commercial pay-per-install on end users across the globe. On an average week, Safe Browsing generates over 60 million warnings related to unwanted software delivered via PPI—three times that of malware. Despite these protections, estimates of unwanted software incident rates provided by the Chrome Cleanup Tool [5] indicate there are tens of millions of installs on user systems. Of the top 15 families installed, we find 14 distribute via commercial PPI.

Thousands of PPI affiliates drive these weekly downloads through a battery of distribution practices. We find 54% of sites that link to PPI bundles host content related to freeware, videos, or software cracks. For the long tail of other sites where users are not expecting an installer, PPI networks provide affiliates with “promotional tools” such as butter bars that warn a user their Flash player is out of date, in turn delivering a PPI bundle. In order to avoid detection by Safe Browsing, affiliates churn through domains every 7 hours or actively cloak against Safe Browsing scans. Our findings illustrate the deceptive behaviors present in the commercial PPI ecosystem and the virulent impact it has on end users.

In summary, we frame our contributions as follows:

- We present the first investigation of commercial PPI’s internal operations and its relation to unwanted software.
- We estimate that commercial PPI drives over 60 million download attempts every week.
- We find that 14 of the top 15 unwanted software families distribute via commercial PPI.
- We show that commercial PPI installers and distributors knowingly attempt to evade user protections.

2 Commercial Pay-Per-Install

For the purposes of this study, we define *commercial pay-per-install* (PPI) as the practice of software developers bundling several third-party applications in return for a fee. We present an example bundle in Figure 1, where clicking on “accept” results in a user installing eight *offers* through a single radio dialogue. Some of these offers may be *unwanted software*, where at least one anti-virus engine on VirusTotal marks the application as potentially unwanted, adware, spyware, or a generic category. In contrast to blackmarket pay-per-install which illegally

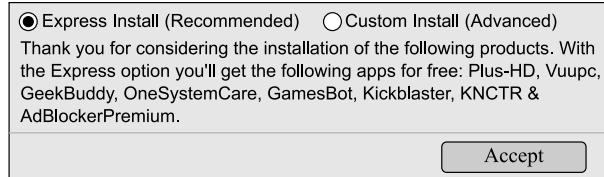


Figure 1: Sample prompt bundling eight commercial pay-per-install offers. Each offer is downloaded and automatically installed upon a user accepting the “Express Install” option. Users may have no knowledge of the behaviors of the bundled offers.

sells access to compromised hosts, deceptive commercial PPI outfits rely on this prompt to nominally satisfy user consent requirements. To simplify the process of buying and selling installs, commercial PPI operates as an affiliate network. We outline this structure and enumerate the major networks in operation during our study.

2.1 PPI Affiliate Structure

The pay-per-install affiliate structure consists of *advertisers*, *publishers*, and *PPI affiliate networks*. Figure 2 presents the typical business role each plays.

Advertiser: In the pay-per-install lingo, advertisers are software owners that pay third-parties to distribute their binaries or extensions. Restrictions on what software advertisers can distribute falls entirely to the discretion of PPI affiliate network operators and their ability (and willingness) to police abuse. As highlighted in Figure 2, advertisers include developers of unwanted software like Conduit, Wajam, or Shopperz that recuperate PPI installation fees by monetizing end users via ad injection, browser settings hijacking, or user tracking. Irrespective of the application’s behavior, PPI networks set a minimum bid price per install that advertisers only pay out upon a successful install. Advertisers may also restrict the geographic regions they bid on.

Publisher: Publishers (e.g., affiliates) are the creators or distributors of popular software applications (irrespective of copyright ownership). An example would be a website hosting VLC player as shown in Figure 2. PPI networks re-wrap a publisher’s application in a *downloader* that installs the original binary in addition to multiple advertiser binaries. This separation of monetization from distribution allows publishers to focus solely on garnering an audience and driving installs through any means. Consequently, advertisers may have no knowledge of the deceptive techniques that publishers employ to obtain installs, nor what their binary is installed alongside. Upon a successful install, the publisher receives a fraction of the advertiser’s bid. We differentiate this from direct distribution licenses such as Java’s agreement to bundle the Ask Toolbar [18], as there is no ambiguity be-

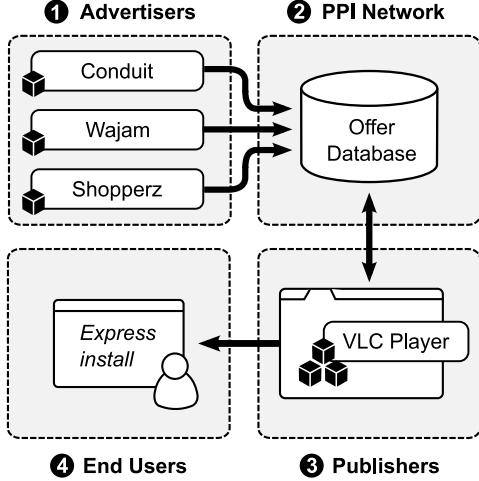


Figure 2: Pay-Per-Install (PPI) business model. Advertisers paying for installs supply their binaries to a PPI affiliate network (1). The PPI network cultivates a set of publishers—affiliates with popular software applications seeking additional monetization (2). The PPI network re-wraps the publisher’s software with a customized downloader that the publisher then distributes (3). When end users launch this downloader, it installs the publisher’s software alongside multiple advertiser binaries (4). The PPI network is paid by the advertisers and the publisher receives a commission.

tween the advertiser and publisher around what packages are co-bundled and the source of installs.

PPI Affiliate Network: PPI affiliate networks serve as a bridge between the specialized roles of advertisers and publishers. The PPI network manages all business relationships with advertisers, provides publishers with custom downloaders, and handles all payments to publishers for successful installs. When a publisher gains access to an end user’s system, the PPI network determines which offers to install. As we show in Section 3, this entails fingerprinting an end user’s system to determine any risk associated with anti-virus as well as to support geo-targeted installations. Similarly, the PPI network dictates the level of user consent when it installs an advertiser’s binary, where consent forms a spectrum between silent installs to opt-out dialogues. In some cases, advertisers can customize the installation dialogue and thus play a role in user consent.

Reselling: With multiple PPI affiliate networks in operation, various PPI operators will aggregate their publishers’ install traffic and resell it to larger PPI affiliate networks. These smaller PPI operators create value for their affiliates by providing promotional tools in the form of landing pages, banner ads, butter bars (e.g., “Your Flash player is out of date”), and generic installers for media players and games—described later in Section 6. These tools simplify the process of monetizing web traf-

PPI Affiliate Network	First Seen	Reseller
AirInstaller	09/2011	
Amonetize	01/2012	
InstallCore	04/2011	
InstallMonetizer	06/2010	
InstallMonster	06/2013	
Installaxy	06/2014	✓
Installerex	12/2013	✓
NetCashRevenue	01/2014	✓
OpenCandy	04/2008	
Outbrowse	11/2012	
PerInstallBucks	06/2013	✓
PerInstallCash	04/2011	✓
Purebits	06/2013	✓
Solimba	08/2013	
Somoto	10/2010	

Table 1: List of 15 PPI affiliate networks, an estimate of when they first started operating, and whether they resell installs.

fic where a victim is not primed to download a bundle. It is worth noting that these resellers do not operate their own downloader; they rely on sub-affiliate tracking provided by larger PPI networks that effectively enables two-tiered affiliate distribution.

2.2 Identifying PPI Networks

In contrast to blackmarket pay-per-install [4], the affiliate networks driving commercial PPI are largely private companies with venture capital backing such as *Install-Monetizer* and *OpenCandy* [8, 9]. Registering as a publisher with these PPI networks is simple: a prospective affiliate submits her name, website, and an estimate of the number of daily installs she can deliver. Given this porous registration process, underground forums contain extensive discussions on dubious distribution techniques and which PPI affiliate networks offer the best conversion rates and payouts. We tracked these conversations on *blackhatworld.com* and *pay-per-install.com*, enumerating over 50 commercial PPI affiliate programs that exclusively deal with Windows installs. While there are networks that target Mac and mobile installs, we focus our work on the relationship between commercial PPI and unwanted software families that disproportionately impact Windows users as identified by previous studies [17, 34].

2.3 Acquiring PPI Downloader Samples

As part of our initial investigation of PPI, we successfully acquired downloaders for fifteen distinct PPI networks. We list each in Table 1. These networks have been in operation for an average of 2–3 years, with the oldest program dating back to 2008 as gleaned from crawl logs provided by *archive.org*. Based on a preliminary black-

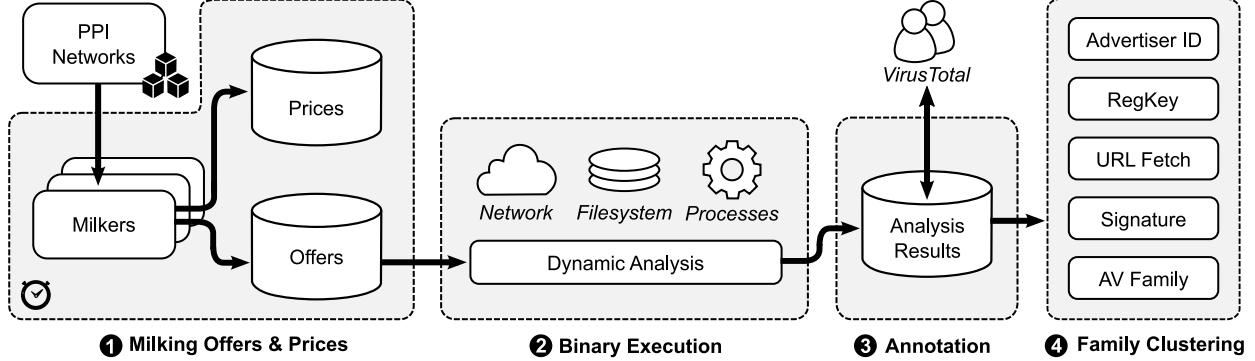


Figure 3: PPI monitoring infrastructure. We collect offers and prices from four PPI networks on a regular basis (❶). We then execute the offer binaries in a sandbox to observe network requests, file system changes, and running processes (❷). We annotate each binary with any known VirusTotal labels (❸) before finally clustering binaries into families (❹).

box test of each downloader, we found six of the fifteen PPI downloaders were merely resellers for other PPI networks in our list. Of the remaining nine, we elect four of the largest—Amonetize, InstallMonetizer, OpenCandy, and Outbrowse—as the basis of our investigation into the role of PPI in unwanted software distribution. We based our initial selection criteria on the complexity of the offer protocols and from preliminary statistics reported by Safe Browsing on which PPI networks delivered the largest number of downloads. We confirm later in Section 5 that these four PPI networks are in fact representative, large operators. We also explore the impact of the PPI ecosystem as a whole on end users.

3 Monitoring the PPI Ecosystem

Using the PPI downloader samples we acquire for Amonetize, InstallMonetizer, OpenCandy, and Outbrowse, we develop a pipeline to track the offers (e.g., advertiser binaries) that each PPI network distributes as well as the regional price per install. We outline our pipeline in Figure 3. We begin by simulating each PPI downloader’s protocol to fetch all possible offers on an hourly basis. We analyze each binary in a sandboxed environment, ultimately clustering the offers into software families based on the behavioral patterns we observe. We discuss the construction of our pipeline and its limitations.

3.1 PPI Downloader Protocol

All four PPI downloaders we study rely on a three-stage protocol for dynamically fetching advertiser binaries. To start, a downloader fingerprints a client’s device to determine the operating system and default browser. The downloader reports these parameters to the PPI server as part of a request for all available offers as shown in Figure 4. In our example, the request embeds the exact

version of the client’s OS and service pack; the Chrome, Firefox, and Internet Explorer version if any are present; whether the system is 32-bit or 64-bit; and finally potentially unique identifiers including a MAC address and a machine identifier such as `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography`.

We provide a typical offer response in Figure 5. Each offer contains a unique product identifier, download URL, and additional metadata that dictates how the install process unfolds. Depending on the PPI network, the response will include anywhere from 5–50 offers, filtered by regional requirements imposed by the PPI server based on the client’s IP address.

In the second stage, the downloader verifies that none of the `RegKey` or `AntivirusRegKeys` are present in the device’s registry. This approach serves to prevent multiple installations of the same advertiser’s binary as well as to avoid anti-virus disrupting the installation of an offer. If a client’s machine satisfies the offer criteria, the downloader will display the offer and execute the binary with the command line options specified by the PPI server if accepted. These parameters sometimes reveal the intent of the advertiser (e.g., replacing the default search provider) as well as evasive actions such as remaining dormant for 20 days to prevent unwanted software symptoms (e.g., injected ads) from manifesting immediately after an installation. If a client’s system does not satisfy the criteria, the downloader simply tests another potential offer until all options are exhausted. In total, the downloader will display offers for anywhere from 1–10 advertiser binaries (potentially all as one express install dialogue): the maximum is dictated by the PPI network.

In the final stage, the downloader reports all successfully installed offers along with the publisher’s affiliate id for compensation.

```
http://srv.desk-top-app.info/Installer/
Flow?os=6.1&ospv=-1&iev=9.11&ffv=&
chromev=46.0&macaddress=00:00:00...
&systembit=32&machineguid=b1420e...
```

Figure 4: Example PPI network request for Outbrowse containing the components that make up a device fingerprint.

```
{
  "SleepAfterInstall": 1800000,
  "ExeURL": "http://example.com/file7",
  "AntivirusesRegKeys": "[
    {"RegKey32": "...\\McAfee..."}],
  "RegKey": ...,
  "PostRegKey": ...,
  "ProductID": 10001,
  "CommandLine": "-defaultsearch=true",
  "RunInAggressiveInstaller": "1",
}
```

Figure 5: Example PPI network response for Outbrowse containing an offer and associated metadata.

3.2 Developing Longitudinal Milkers

In order to track bundled offers, we develop *milkers* that replay the first stage of each PPI network’s offer protocol and decode the response. This is largely a time-intensive manual process of blackbox testing each PPI downloader, determining the PPI server’s domain (or those it cycles through), and re-implementing the protocol into a standalone module that generates a network request with the expected downloader User-Agent and custom headers. We provide a sample of the PPI server domains contacted by downloaders in Table 2.

For all requests, we present a device fingerprint associated with a Windows 7 system with Chrome and Internet Explorer installed while randomizing unique identifiers such as the device’s MAC address and machine ID. Upon receiving a response, we decode the list of offers and extract the associated URL of the offer binary. We reiterate that the PPI programs we monitor provide anywhere from 5–50 potential offers along with their installation requirements. For each offer, we detect whether we previously observed the URL of the associated binary. If the URL is fresh, we download the URL’s content; if the URL is redundant, we rely on a cached copy in order to reduce network load on the PPI servers. We note that this caching methodology may reduce the number of unique digests we obtain if advertisers were to cycle binaries referenced by a fixed URL.

We finally store each binary, the offer metadata (e.g., registry requirements, advertiser ids), and the timestamp of execution. We ran our milkers every hour from a col-

PPI Network	Sample Domain
<i>Outbrowse</i>	srv.desk-top-app.info
<i>Amonetize</i>	www.download-way.com
<i>InstallMonetizer</i>	www.stsunwest.com
<i>OpenCandy</i>	api.opencandy.com

Table 2: Sample of PPI server domains contacted by our milkers. In total, we identify 31 domains servicing offer requests for the four PPI networks we study.

PPI Network	Milking Period	Offers	Unique
<i>Outbrowse</i>	1/08/15–1/07/16	107,595	584
<i>Amonetize</i>	1/08/15–1/07/16	231,327	356
<i>InstallMonetizer</i>	1/11/15–1/07/16	30,349	137
<i>OpenCandy</i>	1/09/15–1/07/16	77,581	134
Total	1/08/15–1/07/16	446,852	1,211

Table 3: Breakdown of PPI networks, milking periods, and the unique offers appearing in our dataset.

lection of cloud instances hosted in the United States over a year long period from January 8, 2015–January 7, 2016. During this time, we updated our milker protocols at most once per PPI network, a reflection of the lack of external pressure on commercial PPI practices compared to malware. In total, we collected 446,852 offers. These offers contained 2,841 unique URLs, 1,809 unique digests, and 1,211 unique product identifiers (as determined by the `ProductID` field shown in Figure 5, or its equivalent for other PPI networks, which are consistent across versions.) We provide a detailed breakdown of the offers per PPI network in Table 3.

We faced a separate challenge for tracking regional pricing. In particular, the exact daily prices that advertisers pay per install are available only to publishers delivering successful installs. Unlike previous investigations into blackmarket PPI [4], we elected not to register as commercial PPI affiliates due to potential Terms of Service violations. As such, we lack access to per-advertiser pricing data. Instead, we track the average price per install across the PPI ecosystem as publicly advertised by PPI networks and resellers to attract affiliates. In total, we identify five PPI-related websites that provide a breakdown of the current price per install paid across 219 regions, with rates varying between \$0.01–\$2.09. These sites include cinstaller.com, installmania.com, cashmylinks.com, perinstallbucks.com, and truemediapartner.com. We crawled and parsed these pages (as allowed by robots.txt) on a weekly basis from January 8, 2015–January 7, 2016 to monitor any fluctuations.

3.3 Executing and Annotating Offers

We execute all downloaded binaries in a sandboxed environment similar in flavor to Anubis [2], CWSandbox [38], and GQ [22], the details of which are covered in previous work [17, 29, 34]. During execution, we log all network requests and responses, file system changes, modified registry keys, and spawned processes. We also monitor whether the executable changes any preferences related to Chrome or Internet Explorer such as altering the default browser, dropping an extension, or modifying the startup page.

Independent of our dynamic execution environment, we annotate each binary with third-party intelligence gathered through VirusTotal at the end of our collection period. Mechanically, we submit the hash of each binary to determine which of 61 anti-virus engines report the binary as malicious or unwanted. We also collect any labels, though the value of these is highly variable: some reflect generic ‘Adware’ while others contain a family name potentially unique to an anti-virus engine.

3.4 Clustering and Classifying Offers

At the conclusion of our collection period we classified all of the advertiser binaries in our dataset into distinct families. This canonicalization step is necessary to de-duplicate instances where the same advertiser works with multiple PPI networks or where advertisers introduce polymorphism due to software updates, sub-affiliate programs, or to evade detection by anti-virus engines. Classification is a semi-automated process where we first cluster all binaries based on overlapping registry key modifications, domains contacted during execution, process names, or digital certificates used to sign the advertiser’s software (only 58% of offers were signed). This approach follows similar strategies for clustering malware delivered via drive-by downloads [14] and unwanted software using code-signing [21]. We also cluster offers based on the registry keys present in the installation pre-conditions provided by PPI networks during offer selection. These pre-conditions unambiguously reveal all of the registry paths controlled by a single family, such as *Vitruvian* which goes by 19 other names including *LessTabs*, *SearchSnacks*, *Linksicle*; or *Wajam* which installs under 418 unique registry keys. We present a sample of these pre-conditions in Figure 6. Through all these clustering techniques, we generate 873 non-overlapping clusters (of 1,809 possible).

We manually review all clusters active for more than 150 days (e.g., we examine the timestamp of all milked binaries in a cluster and count the number of distinct dates) totaling 58 distinct clusters. We derive family labels based on the most common naming convention

```
this.bCompExist = g.ami.CheckRegKey(
    "Software\\Wajam",
    "Software\\WInternetEnhance",
    "Software\\WajNEnhance",
    "Software\\WWebEnhance",
    "Software\\WaWebEnhance",
    "Software\\WajIntEnhancer",
    "Software\\WajaIntEnhancer",
    "Software\\WNEenhancer",
    "Software\\WajaInternetEnhance",
    "Software\\WInterEnhance",
    "Software\\WajNetworkEnhance",
    "Software\\WajaNetworkEnhance",
    "Software\\WWebEnhancer",
    "Software\\WaWebEnhancer",
    "Software\\WajWebEnhancer",
    "Software\\WajaWebEnhancer",
    ....
    "Software\\Wajam\\affiliate_id")
```

Figure 6: Example offer requirements for Wajam via Amonetize. It contains 418 registry key checks for Wajam variants. We cluster offers that contain the same registry checks.

found in VirusTotal for a cluster. If no public name exists, we fall back to the advertiser name listed in the offer metadata provided by PPI networks. For all clusters lasting less than 150 days, we rely exclusively on the advertiser name. These names serve only to communicate the major software families commonly found in commercial PPI and whether they overlap with the largest unwanted families impacting end users (discussed in Section 5).

3.5 Limitations

Our investigation of the PPI ecosystem faces a number of limitations. First, our pipeline runs exclusively from United States IP addresses. This potentially biases our perspective of PPI offers in the event advertisers distribute exclusively to non-US territories. As we demonstrate later in Section 4, the US is the highest paid region for installs, which makes it one of the most interesting to analyze. Next, because we do not participate directly in the PPI ecosystem, we lack exact pricing details per install. We attempt to extrapolate these values based on public pricing used to attract affiliates, but we cannot verify the accuracy of this data other than to corroborate similar rates cited within the underground. Third, our family classification faces the same challenges of malware phylogeny where there is frequent disagreement between anti-virus naming conventions. We reconcile these discrepancies for the longest running PPI campaigns at the expense of overlooking the long tail of brief campaigns. Finally, our perspective of the PPI ecosystem is restricted to four PPI networks due to the time-intensive

process of building milkers. While there is a risk our findings are not representative of the entire ecosystem, we show in Section 4 there is substantial overlap between the advertisers of each PPI network. This leads us to believe our sample of PPI networks extends to other unexplored commercial PPI operators.

4 Exploring Commercial PPI Offers

We provide a bird’s-eye-view of the business relationships underpinning the commercial PPI ecosystem before diving into the unwanted software families reliant on PPI distribution. We find that ad injectors, browser settings hijackers, and system “clean-up” utilities dominate the advertisers paying for installs. With anti-virus engines flagging 59% of the weekly software families we milk per PPI network, we observe at least 20% of PPI advertisers take advantage of anti-virus and VM detection provided by PPI downloaders to avoid installing in hostile environments.

4.1 High-Level Metrics

Using the 1,211 product identifiers embedded by PPI networks in each offer for accounting purposes, we calculate the total distinct simultaneous offers per PPI network and the duration that advertisers run each offer. On average, we observe 25–60 active offers per PPI network each week, with a fine grained breakdown shown in Figure 7. The spike around July 2015 for Amonetize represents a temporary 2x increase in offers distributed by the PPI network; it is unrelated to any change in our infrastructure. The majority of advertisers for Amonetize and Outbrowse maintain their offers for less than a week before cycling to a new product as shown in Figure 8. In contrast, OpenCandy and InstallMonetizer attract advertisers who run the same product for over 15 days.

4.2 Longest Running Campaigns

With over 873 software families classified by our analysis pipeline, we examine which families consistently appear in the PPI ecosystem and thus sink the most money into installs. Table 4 provides a detailed breakdown of the software families with the longest running distribution campaigns and the PPI networks involved. The families fall into five categories: ad injectors, browser settings hijackers, system utilities, anti-virus, and major brands. We provide sample screenshots of the resulting user experience after installation in the Appendix.

Ad Injectors: Ad injectors modify a user’s browsing experience to replace or insert additional advertisements that otherwise would not appear on a website. Every PPI network we monitor participates in the distribution of ad injectors. Of the top eight programs listed by Thomas

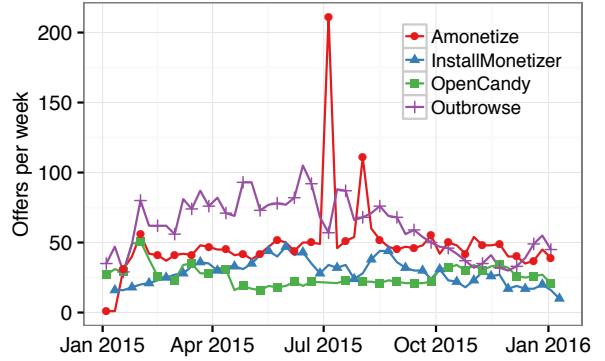


Figure 7: Unique PPI offers operating each week. Amonetize and Outbrowse cultivate a large number of offers compared to OpenCandy and InstallMonetizer.

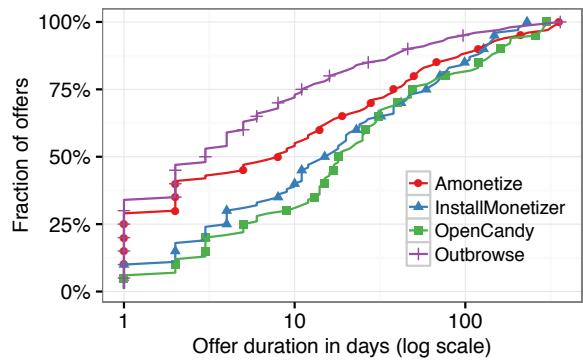


Figure 8: Lifetime of PPI offers. Advertisers run the same offer on OpenCandy and InstallMonetizer for a median of 15 days, while Amonetize and Outbrowse offers quickly churn out of existence to be replaced by new binaries.

et al. as the largest contributors to ad injection in 2014 for Chrome, Firefox, and Internet Explorer [34], we observe six currently in the PPI ecosystem. The companies behind these software products are commercial entities that span the globe: *Wajam* is located in Canada, *Eorezo* is from France, while *Crossrider* originates from Israel. These ad injectors recuperate the initial sunk cost of installs by monetizing users via display ads and shopping helpers until a victim finally uninstalls the injector.

Browser Settings Hijackers: Settings hijackers modify a victim’s default browser behavior, typically to change the default tab or search engine to a property controlled by the hijacker. These companies subsequently monetize victims by selling their traffic to search engines and potentially tracking user behavior. Examples include Conduit Search (e.g., Search Protect) which came pre-installed on Lenovo machines in 2014 [3]. We note that some hijackers also profit by doubling as ad injectors.

System Utilities: System utilities attempt to upsell users using potentially deceptive practices, with some meeting anti-virus definitions of scareware. This category includes “speedup” utilities like *Speedchecker* and *Uniblue* that present nebulous claims such as “Attention! 2203 items are slowing down your PC” or “your system registry health status is dangerous.” These families repeatedly generate pop-up warnings until a victim either pays a subscription fee of \$30–40 or uninstalls the software. This scheme is nearly identical to fake anti-virus, but speedup utilities operate under a veil of legitimacy because they remove files from a client’s machine, thus satisfying some notion of system improvement. Consequently, anti-virus engines do not consider these families to be malicious, only unwanted. Our categorization also includes cloud backup utilities that repeatedly prompt victims to upload their files to the cloud. Adhering to the dialogue requires victims pay a recurring \$120 subscription fee.

All five of the top system utility families are themselves affiliate programs. *Speedchecker* promises affiliates a 30% commission on subscriptions. *Uniblue* advertises a commission of 70%. What emerges is a three-tiered distribution network where system utility affiliates register as advertisers on PPI networks and pay an up-front distribution cost, but reap the commissions on successful subscription conversions. It is also possible that the system utility companies maintain a direct relationship with PPI networks.

Anti-Virus: We observe four anti-virus products distributed via the PPI ecosystem: AVG, LavaSoft, Comodo, and Qihoo. We cannot determine whether these companies directly purchase installs from commercial PPI affiliate networks. We note that all four operate affiliate programs to outsource installs [1, 7, 24, 37]. Assuming all of the installs we observed originate from affiliates, it is unclear how each anti-virus operator polices abuse in the face of an increasingly tangled web of purchased installs and potentially dubious distribution practices. Equally problematic, PPI downloaders simultaneously install these anti-virus products alongside browser settings hijackers and ad injectors—an unenviable user experience.

Major Brands: We observe a small number of major software brands including *Opera*, *Skype*, and browser toolbars distributed via PPI. Based on the affiliate codes embedded in the download URLs for *Opera*, it appears that *Opera* directly interacts with PPI operators to purchase installs rather than relying on intermediate affiliates.¹ The other three programs all operate affiliate pro-

Category	Family	Days	Networks	AV
Ad Injector	<i>Wajam</i>	365	A, C, I, O	13
Ad Injector	<i>Vopackage</i>	365	A, I, O	42
Ad Injector	<i>Youtube Downloader</i>	365	A, I, O	50
Ad Injector	<i>Eorezo</i>	365	A, O	32
Ad Injector	<i>Crossrider</i>	350	A, I, O	55
Ad Injector	<i>Bubble Dock</i>	340	O	8
Ad Injector	<i>Nuvision Remarketer</i>	322	A	18
Ad Injector	<i>Download Manager</i>	313	A	37
Ad Injector	<i>Vitruvian</i>	242	A, I, O	41
Hijacking	<i>Browsefox</i>	363	A, C, I, O	49
Hijacking	<i>Conduit</i>	327	A, I, O	41
Hijacking	<i>CouponMarvel</i>	300	A	3
Hijacking	<i>Smartbar</i>	294	A, I, O	45
Hijacking	<i>Safer Browser</i>	279	A, I, O	3
Utilities	<i>Speedchecker</i>	365	A, O	5
Utilities	<i>Uniblue</i>	347	A, C, I, O	49
Utilities	<i>OptimizerPro</i>	302	A, C, I, O	29
Utilities	<i>My PC Backup</i>	292	A, C, I	2
Utilities	<i>Pro PC Cleaner</i>	287	A, I, O	33
Utilities	<i>Systweak</i>	249	A, I, O	37
Anti-virus	<i>AVG Toolbar</i>	333	A, C	0
Anti-virus	<i>LavaSoft Ad-aware</i>	305	C	0
Anti-virus	<i>Comodo GeekBuddy</i>	153	A, C, I, O	0
Anti-virus	<i>Qihoo 360</i>	144	C, I	0
Brand	<i>Opera</i>	340	A, C, I, O	0
Brand	<i>Skype</i>	176	C, O	0
Brand	<i>Yahoo Toolbar</i>	27	O	5
Brand	<i>Aol Toolbar</i>	25	O	4

Table 4: Software families with the longest PPI campaigns. We annotate each with the type of software, the days the campaign ran for, the PPI networks involved, and the number of anti-virus engines that flag the family as unwanted. We abbreviate PPI networks as [A]monetize, Open[C]andy, [I]nstallMonetizer, and [O]utbrowse.

grams, yielding a similar distribution pattern to that of anti-virus, though we cannot rule out direct relationships with commercial PPI.

4.3 Long Tail of Campaigns

Outside the top 28 longest running PPI campaigns, a question remains on the mixture of credible and unwanted software that makes up the other 845 short lived campaigns. To explore this, we calculate the fraction of software families distributed per week by commercial PPI where at least one anti-virus engine in VirusTotal flags the family as unwanted. Figure 9 presents our results. On an average week, anti-virus engines label 85% of software families distributed by InstallMonetizer

¹For example, we observe Outbrowse specifically referenced in the target download URL for *Opera*: net.geo.opera.com/opera/

stable?utm_medium=pb&utm_source=outbrowse&utm_campaign=2328

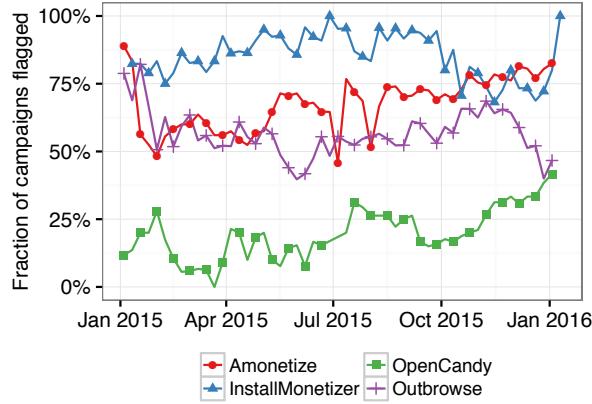


Figure 9: Fraction of software families found each week in PPI networks that were flagged by any anti-virus engine in VirusTotal.

as unwanted, compared to 68% for Amonetize, 57% for Outbrowse, and 20% for OpenCandy. These trends hold true for the entirety of our year-long monitoring. Our findings illustrate that unwanted software dominates both long and short-lived campaigns. The only exception is OpenCandy, which predominantly cultivates advertisers related to games and anti-virus, and to a lesser extent, system utilities and some ad injectors. As a consequence though, OpenCandy has the smallest pool of offers (as discussed previously in Figure 7), while other PPI networks deal with a large number of unwanted software creators and affiliates.

4.4 Contending with Anti-Virus

As discussed in Section 3, each PPI network provides advertisers with a capability to pre-check whether an anti-virus engine is present prior to displaying the advertiser's offer. This pre-check consists of a blacklist of registry keys, file paths, and registry strings specified by the advertiser. We present a sample in Figure 10. To estimate the fraction of offers that take advantage of this capability, we manually collate a list of 58 common anti-virus tokens that appear in a random sample of pre-check requirements, as well as the names of anti-virus companies participating in VirusTotal. We then scanned all offer installation requirements for these tokens.

Of the unique offers in our dataset, 20% take advantage of PPI downloader capabilities that prevent installs from occurring on clients running an anti-virus engine. When anti-virus checks are present, we find advertisers target an average of 3.6 AV families. Our findings suggest that PPI networks support unwanted software developers as first-class partners. We caution our metric is a strict underestimate in the event PPI download-

```

g_ami.CheckRegKey(
    "Software\\Avast Software"
    "Software\\Symantec"
    "Software\\KasperskyLAB"
    "Software\\Norton"
    "Software\\Microsoft\\Microsoft Anti.."
    "Software\\Microsoft\\Microsoft Secu.."
    "Software\\Malwarebytes"
    "Software\\Avira")
g_ami.PathExists(
    "%ProgramFiles%\\mcafee"
    "%ProgramFiles%\\Microsoft Security..."
    "%ProgramFiles%\\Malwarebytes...")

```

Figure 10: Example of anti-virus checks performed by a PPI downloader in order to avoid displaying certain offers to clients running hostile anti-virus engines.

ers scan for side-effects related to anti-virus rather than the exact brand names. We find the most frequently targeted brands include ESET, Avast, AVG, McAfee, Avira, and Symantec. We also observe offers checking for registry keys related to VirtualBox, VMWare, and OpenVPN. There are two possible interpretations of this behavior: advertisers seek to protect themselves from fraudulent installs on virtualized systems; or advertisers actively prevent installations on suspected security testing environments. Given the virtualization checks co-occur with anti-virus evasion, we hypothesize the latter is more likely. Added to our earlier observation that PPI downloaders provide a capability to impose a symptom-free quiet period after installation, a picture emerges of PPI networks actively supporting unwanted software as a first-class partner.

4.5 Regional Pricing Per Install

Far and away, installs from the United States fetch the highest price at roughly \$1.50 each. The United Kingdom is the second most lucrative region at roughly \$0.80 per install. We find that advertisers pay the highest rates for installs from North America, Western Europe, and Japan as shown in Figure 11. Prices outside these regions hover around \$0.02–\$0.10 per install. This holds true throughout the entirety of our investigation as shown in Figure 12 with relatively little volatility in the market. Despite these lower rates, we show in the next Section that commercial PPI impacts clients around the globe.

5 Measuring User Impact

Through Safe Browsing, we estimate the virulent impact that the PPI ecosystem has on end users. Beginning in 2014, Safe Browsing added support to warn users of Chrome and Firefox against downloading PPI-laden

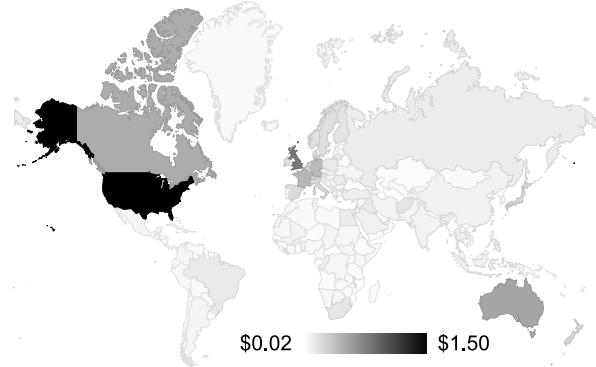


Figure 11: Average price per install across all PPI price monitoring vantage points. Installs from the United States fetch the highest price at \$1.50 each.

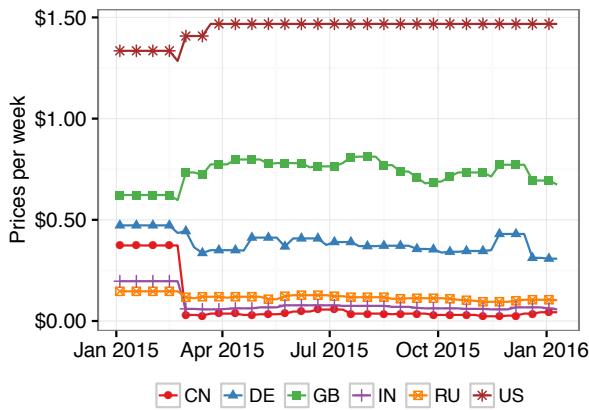


Figure 12: Weekly average price per install for six regions. We observe relatively little volatility for US installs and a slight decline in rates in Europe over time.

software that violates Google’s unwanted software policy [28]. This policy covers a subset of applications flagged by anti-virus engines as unwanted. We map these metrics to the PPI networks we study and find that Safe Browsing generates over 60 million weekly download warnings and browser interstitials. Despite these protections, telemetry Chrome users submit about their systems indicate there are tens of millions of installations of unwanted software, with nearly all of the top families contemporaneously paying for installs.

5.1 Requests for PPI Downloader

We rely on two datasets to estimate the volume of weekly downloads to software monetizing through Amonetize, InstallMonetizer, OpenCandy, and Outbrowse: (1) pings reported by browsers integrated with Safe Browsing for downloaded binaries; and (2) Safe Browsing’s repository of over 75 million binaries (including benign software). When a browser integrated with Safe Browsing fetches a binary from an untrusted source, it generates an API re-

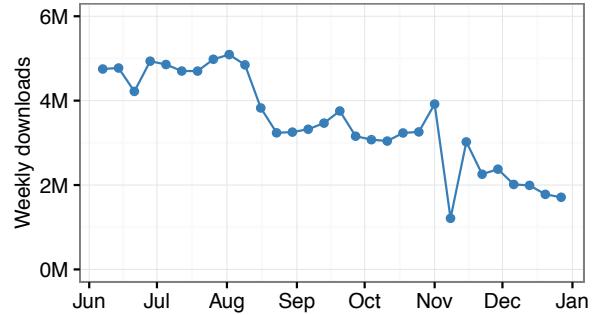


Figure 13: Volume of weekly requests for any of 1.5 million PPI downloaders. We stress this is a lower bound due to missing samples.

quest to Google in order to obtain a verdict for whether the binary is unwanted or malicious. This request contains hosting details about the binary (e.g., URL, IP address) and related metadata including a digest of the binary [27]. In order to map these downloads to digests of known PPI downloaders, we scan Safe Browsing’s repository of dynamic execution traces in search of network requests that match the offer discovery protocol used by each PPI affiliate network (previously discussed in Section 3). From this repository, we identify 1.5 million binaries tied to one of the four PPI networks we study.

We show the total weekly downloads for these 1.5 million binaries between June 1, 2015–January 7, 2016 in Figure 13, irrespective of whether Safe Browsing displayed a warning. We caution these estimates of traffic to PPI networks should serve only as a lower bound as Safe Browsing’s coverage of all possible binaries is incomplete. Similarly, due to Safe Browsing displaying warnings for policy-violating PPI downloaders, operators have an incentive to quickly cycle binaries and hosting pages. Caveats aside, we find publishers for the four PPI networks drive an average of 3.5 million downloads per week, though the volume appears to be in decline. Even as a lower bound, our results illustrate the massive influence that PPI networks have on unwanted software distribution.

5.2 PPI Downloader Warnings

In order to obtain a broader perspective of the entire PPI ecosystem’s impact on end users (not just the four networks we study), we measure the volume of weekly warnings generated by Safe Browsing for PPI downloaders. Users encounter warnings in one of two ways: download warnings that trigger for policy-violating PPI downloaders, and full-page interstitials that appear when users visit websites commonly distributing PPI-laden software. Because affiliate publishers attempt to evade detection (discussed more in Section 6), Safe Browsing

relies on a reputation system called CAMP that builds on incomplete data [29]. The system starts from a seed set of 3 million PPI downloaders that includes samples for all fifteen PPI networks we outlined previously in Section 2. From there, the system scores websites hosting these binaries, common redirect paths, and related binaries. This expands the coverage of sites and binaries involved in pay-per-install, but results in a loss of attribution to individual PPI families. As such, we can only provide an aggregate impact estimate.

We present the volume of PPI downloader warnings and page-level interstitials generated by Safe Browsing between June 1, 2015–January 7, 2016 in Figure 14 and Figure 15 respectively. On an average week, Safe Browsing raises 35 million download warnings and displays 28 million interstitials. Warnings appear as a bursty process, in part due to the arrival of new distribution campaigns and in part due to the ongoing evolutions in the reputation of websites and binaries. In order to place unwanted software in the greater context of threats facing users, we compare the volume of users encountering PPI downloaders versus malware. On average, Safe Browsing raises 13.5 million download warnings and 9 million interstitials to protect users from malware—three times less than that of unwanted software.

The risk of unwanted software is not localized to any single region. We provide a breakdown of the geolocation of users shown a warning related to PPI downloaders in Table 5. We find that Indian users account for 8% of warnings, followed in popularity by Brazil, Vietnam, and the United States. We find no correlation between the price per install and geographic regions with high incident rates. As such, it appears that PPI networks drive installs to any possible user, even when the payout hovers around \$0.10 per install.

5.3 Existing Unwanted Installs

For those PPI downloaders that escape detection and launch on a client’s machine, we estimate the number of users potentially affected. To do this, we tap into metrics kept by the Chrome Cleanup Tool, an opt-in tool that scans a user’s machine for symptoms induced by popular ad injectors, browsing settings hijackers, and system utilities and removes offending programs [5]. Given hundreds of potential unwanted software strains, the tool prioritizes families based on telemetry built into Chrome and system traces supplied by users who file Chrome complaints due to undesirable user experiences. As part of its execution, the tool reports which unwanted software families it identifies as well as those successfully removed. One limitation with the tool is that, for privacy reasons, no unique device identifier is reported per execution. Consequently, if the tool fails to remove a un-

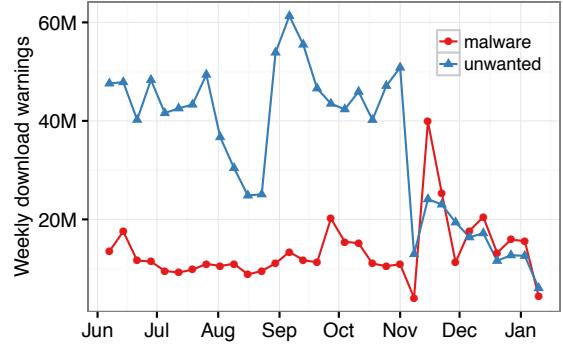


Figure 14: Breakdown of weekly download warnings displayed by Safe Browsing for unwanted software compared to malware. The bursty behavior results from evasion on the part of PPI publishers.

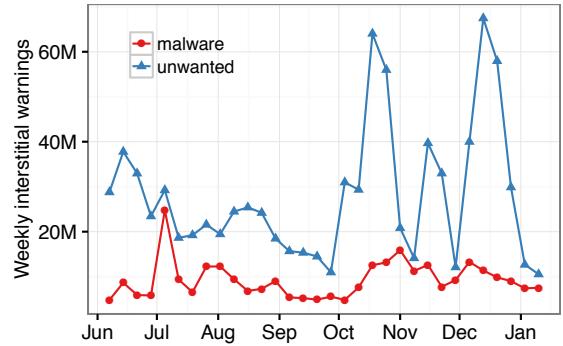


Figure 15: Breakdown of weekly page-level interstitials displayed by Safe Browsing for unwanted software compared to malware. The bursty behavior results from evasion on the part of PPI publishers.

wanted software strain and a user re-runs the tool, they will be counted twice. This may cause us to overestimate the number of infections per family. As the Chrome Cleanup Tool is opt-in, we caution its metrics cover only a subset of all infected machines. While this precludes absolute estimates on the number of unwanted software installs, we can still estimate the relative population of each software family.

Over the last year, the Chrome Cleanup Tool identified *tens of millions* of installations of unwanted software. We present the top 15 most popular strains flagged from January 8, 2015–January 7, 2016 in Table 6. We measure popularity as the total installs per family divided by all known unwanted software installs. To map these families back to the PPI ecosystem, we mark each family known to distribute via any of the four PPI networks we monitor. We arrive at this determination by running the Chrome Cleanup Tool at the completion of the binary execution phase of our analysis pipeline (described in Section 3) to see whether the tool flagged any of the binary’s components.

Country	Frac. Downloads	Price per install
India	8.2%	\$0.09
Brazil	7.2%	\$0.13
Vietnam	6.4%	\$0.06
United States	6.2%	\$1.50
Turkey	5.1%	\$0.11
Thailand	3.3%	\$0.11
Pakistan	3.2%	\$0.08
Mexico	2.6%	\$0.07
Indonesia	2.5%	\$0.09
Philippines	2.5%	\$0.08

Table 5: Top 10 countries receiving the largest volume of Safe Browsing warnings related to unwanted software.

Our results indicate that 14 of the top 15 software families flagged by the Chrome Cleanup Tool simultaneously pay for installs during our monitoring. Conduit, the top family, is a browser settings hijacker that accounts for 20.9% of all unwanted software installs reported by the Chrome Cleanup Tool. Multiplug, the most popular ad injector, accounts for 5.1% of installs. Our results illustrate the virulent affect that unwanted software found in the PPI ecosystem has on end users. We caution we cannot definitively say all of these installs stem from PPI²; there are potentially other sources of installs such as direct downloads via deceptive websites and advertising. However, paired with millions of Safe Browsing warnings for PPI downloaders, we argue that PPI plays a substantial role in unwanted software installation levels.

6 Distribution Techniques

We conclude our investigation with an examination of the affiliates responsible for distributing PPI downloaders, the landing pages they operate, and the deceptive practices that they employ to drive installs.

6.1 Estimating PPI Affiliates

We estimate the number of affiliates participating in Amonetize, InstallMonetizer, OpenCandy, and Outbrowse by scanning for publisher identifiers that each PPI downloader embeds in offer requests for accounting purposes. Based on the dynamic traces of roughly 1.5 million PPI downloaders provided by Safe Browsing (discussed previously in Section 5), we estimate there are 2,518 publishers in the ecosystem, some of which may participate in multiple PPI networks and thus should not be considered unique. We provide a breakdown per PPI network in Table 7. Drawing these estimates into the

²Once a PPI downloader executes, only symptoms related to the bundled advertiser software subsist after the installer completes. The Chrome Cleanup Tool cannot provide us any details for whether unwanted software originated from a PPI downloader.

Unwanted Family	Popularity	PPI Advertiser
<i>Conduit</i>	20.9%	✓
<i>Elex</i>	13.4%	✓
<i>Multiplug</i>	5.1%	✓
<i>Crossrider</i>	4.6%	✓
<i>Browsefox</i>	3.8%	✓
<i>My PC Backup</i>	2.8%	✓
<i>Systweak</i>	2.8%	✓
<i>Mobogenie</i>	2.4%	✓
<i>Smartbar</i>	2.2%	✓
<i>Wajam</i>	1.8%	✓
<i>AnyProtect</i>	1.7%	✓
<i>WinZipper</i>	1.5%	✗
<i>Vopackage</i>	1.2%	✓
<i>ShopperPro</i>	1.2%	✓
<i>Vitruvian</i>	1.1%	✓
Other families	33.5%	—

Table 6: Top 15 software families as detected by the Chrome Cleanup Tool on Windows systems. Popularity is the fraction of all known unwanted software installs.

broader context of PPI, we find a relatively small ecosystem that consists of hundreds of advertisers paying for unwanted software installs that a few thousand publishers distribute. Despite the low number of actors in the space, the end result is still millions of unwanted download attempts on a weekly basis.

6.2 Landing Pages

In order to drive installs, PPI affiliates must present content that either entices or deceives a victim into downloading and executing a PPI downloader. We obtain a sample of these *landing pages* from Safe Browsing which monitors the entire redirect chain behind unwanted software delivery [27]. However, for privacy reasons, our analysis is restricted to a two week period after which these fine-grained details disappear. In total, we sample the top 15,000 most visited landing pages from January 18–February 1, 2016 that direct to one of the four PPI downloaders we monitor. The sites topping this list include large software companies like *utorrent.com*, *bittorrent.com*, and *savefrom.com* (a YouTube downloading service); download portals like *filehippo.com*; and video and media torrent sites like *thepiratebay.se* that display deceptive ads that in fact link to PPI downloaders.

In order to gain a perspective of the category of sites involved in PPI distribution, we crawl all of the landing pages in our sample and supply the non-HTML formatted text to a topic modeling algorithm similar to Gensim’s implementation of LDA [13]. We present the top 10 topics in Table 8, covering 53.6% of all sampled land-

PPI Network	Binary Samples	Affiliates
<i>Outbrowse</i>	1,182,910	1,106
<i>Amonetize</i>	237,660	420
<i>OpenCandy</i>	43,677	747
<i>InstallMonetizer</i>	22,879	245
Total	1,487,126	2,518

Table 7: Estimate of unique affiliates per PPI network. These affiliates drive millions of weekly downloads to PPI networks.

ing pages. Users searching for freeware, video games, torrents, cracks, and even anti-virus are highly likely to encounter PPI downloaders. Most of these sites (58%) cater to an English audience, followed in popularity by Russian (10%). Our results illustrate that popular download portals (or their contributors) fuel a large segment of unwanted software distribution, in turn receiving a kick-back from PPI networks.

6.3 Distribution Pages

After a victim engages with a landing page, PPI affiliates redirect the victim to a *distribution page* that hosts the PPI downloader. This site may be operated by the affiliate or directly by the PPI network, with flavors varying per PPI network. We find that PPI operators rapidly churn through distribution pages, likely to avoid unwanted software warnings from Safe Browsing due to an increasingly negative reputation. During the eight months from June 1, 2015–January 7, 2016, we observed 191,372 distribution pages involved in hosting PPI downloaders. We estimate the lifetime of these pages by measuring the time between the first client that reports a download attempt to Safe Browsing and the last reported download attempt, irrespective of Safe Browsing raising a warning. We find the median lifetime of an Amonetize distribution page is 7 hours, compared to 0.75 hours for Outbrowse. These two stand in contrast to InstallMonetizer and OpenCandy, where distribution pages remain operational for a median of 152 days and 220 days (the entire monitoring window) respectively. This longer lifetime results in part from Safe Browsing not warning on all OpenCandy installs as they do not fall under Google’s unwanted software policy, and in part due to Outbrowse and Amonetize controlling distribution pages, simplifying the process of churning through domains.

6.4 Evasion & Cloaking

Even if PPI operators rapidly cycle through distribution pages, there is a risk that Safe Browsing will scan and detect the PPI downloader itself. We find anecdotal evidence that PPI networks work to actively evade this scanning process. For example, when Safe Browsing first launched its unwanted software detection, it cov-

Site Category	Fraction of Sites
Freeware & Shareware	11.8%
Video Games	10.6%
File Sharing & Hosting	7.3%
Online Video	7.0%
Operating Systems	4.3%
Mobile Apps & Add-Ons	3.7%
Hacking & Cracking	2.7%
Photo & Video Software	2.3%
Game Cheats & Hints	2.1%
Antivirus & Malware	1.9%
Other	46.4%

Table 8: Categorization of the top 15,000 pages driving traffic to PPI downloaders based on topic modeling.

ered only executable files. Shortly after, PPI networks switched to distributing .zip compressed binaries to avoid scanning. When Safe Browsing expanded its scanning coverage, PPI networks moved to more esoteric compression formats including .rar and .ace or doubly compressed files. We also observed PPI networks exploiting a limitation in Chrome, where files downloaded through Flash were not subject to Safe Browsing scans. After a recent Chrome patch to address this, PPI networks switched to password protecting their compressed files, providing instructions for victims on how to access the contents. We provide screenshots of each of these techniques in action in the Appendix. This arms race illustrates that PPI networks opt to actively circumvent user protections rather than ceasing to distribute harmful unwanted software. This behavior likely stems from an incentive structure within PPI where remaining profitable entails racing to the bottom of deceptive install tactics.

6.5 Promotional Tools

For affiliates that do not operate download portals or peer to peer sharing sites, PPI resellers provide deceptive “promotional tools” that socially engineer web visitors into running PPI downloaders. These tools fall into four flavors: butterbars, ad banners, landing pages, and content unlockers.

Butterbars: PPI resellers like NetCashRevenue provide a JavaScript stub to website operators that generates a yellow bar at the top of a page alerting a victim that their “Flash player is out of date!”. This bar can either initiate an auto-download upon visiting the page, or require a victim to click. Either way, the victim receives a PPI downloader.

Content Lockers: Content lockers present victims with an enticing video, song, or PDF. In order to view this content however, a victim must first install a “codec” that

is in fact a PPI downloader. Resellers simplify this process by providing a drop-in script that handles spoofing a fake video player and codec alert.

Ad banners & Landing Pages: Resellers will provide webmasters with ad banners or entire customized landing pages that spoof popular software downloads including uTorrent, Java, Flash, and Firefox that are in fact PPI downloaders.

These techniques highlight that even if the software delivered by a PPI downloader appears benign, the distribution practices of affiliates add an additional layer into the determination of whether software is ultimately unwanted. Consequently, advertisers, publishers, and PPI networks all bear responsibility for the current state of commercial pay-per-install and its ties to unwanted software.

7 Related Work

Blackmarket Pay-Per-Install: Our work is influenced in part by prior explorations of the blackmarket pay-per-install ecosystem that sells access to compromised hosts. Industry reports initially qualitatively described these underground markets as early as 2009 [10,33]. Caballero *et al.* performed the first in-depth investigation by infiltrating the markets and tracking the malware families paying for installs [4]. Prices per install ranged from \$0.02–\$0.18, an order of magnitude less than the prices we observed for commercial PPI. These low rates make blackmarket PPI a better bargain for malware distribution over commercial PPI, though evidence exists of cross-over, such as the commercial PPI network iBarrio recently distributing Sefnit [35]. Other studies have explored the relationships between blackmarket PPI networks and particular malware families [23, 30]. However, all of these studies were limited to establishing a link between the most notorious malware families and their simultaneous distribution in blackmarket PPI; none determined whether PPI was the primary distribution mechanism (as opposed to social engineering or drive-by). Our study went one step further, establishing the volume of weekly download attempts to commercial PPI downloaders.

Unwanted Software: Unwanted software is not a new threat. In 2004, Saroiu *et al.* found at least 5% of computers connected to the University of Washington’s campus network were infected with some form of spyware [32]. In 2005, Edelman tracked multiple purported spyware and adware companies including Claria, WhenU, and 180Solutions to identify their deceptive installation methods and their monetization model [11, 12]. More recently, Thomas *et al.* found that 5% of unique IPs accessing Google websites exhibited symptoms of

ad injection [34], while Jagpal *et al.* identified millions of browsers laden unwanted extensions performing ad injection, search hijacking, and user tracking [17]. Researchers have also explored some of the distribution techniques involved. In 2006, Moshchuk *et al.* crawled and analyzed 21,200 executables from the Internet and found 13.4% contained spyware [25]. Kammerstetter *et al.* repeated a similar study limited to sites purportedly hosting cracks and key generators, though they found the majority bundled malware, not unwanted software [19]. Our work explored the commercialization of these distribution practices as simplified by commercial pay-per-install affiliate networks.

More recently, Kotzias *et al.* explored code-signing techniques of unwanted software that may lead to reduced detection [21]. We rely on a similar technique for clustering advertiser binaries, though we note that only 58% of the 1,809 unique offer digests we identified contained a signature; similarly, only 50% of 1.5 million PPI downloaders distributed by publishers contained a signature. This may lead to a bias in analysis that focus solely on signed unwanted software. Contemporaneous with our own study, Kotzias *et al.* explored the download graph of unwanted software via Symantec’s WINE database and identified 54% of users were affected by unwanted software [20]. Similarly, Nelms *et al.* explored the role of deceptive advertising in enticing victims into running PPI downloaders [26]. Combined with our own work, these three studies present a broad perspective of the number of users affected by unwanted software, an insider perspective of how advertisers, affiliate networks, and publishers coordinate, and the deceptive practices used to entice downloads via advertisements or free software sites.

8 Conclusion

Our work presented the first deep dive into the business practices underpinning the commercial pay-per-install ecosystem that sells access to user systems for prices ranging from \$0.10–\$1.50 per install. Our study illustrated that PPI affiliate networks supported and distributed unwanted software ranging from ad injectors, browser settings hijackers, and system utilities—many of the top families that victims proactively purge from their machines with the aid of the Chrome Cleanup Tool. In aggregate, the PPI ecosystem drove over 60 million weekly download attempts, with tens of million installs detected in the last year. As anti-virus and browsers move to integrate signatures of unwanted software into their malware removal tools and warning systems, we showed evidence that commercial PPI networks actively attempted to evade user protections in order to sustain their business model. These practices demonstrate that

PPI affiliate networks operated with impunity towards the interests of users, relying on a user consent dialogue to justify their actions—though their behaviors may have changed since the conclusion of our study. We hope that by documenting these behaviors the security community will recognize unwanted software as a major threat—one that affects three times as many users as malware.

In response to deceptive behaviors within the commercial PPI ecosystem, members of the anti-virus industry, software platforms, and parties profiting from commercial PPI have formed the Clean Software Alliance [6]. The consortium aims to “champions sustainable, consumer-friendly practices within the software distribution ecosystem.” This includes defining industry standards around deceptive web advertisements, user consent, software functionality disclosure, and software uninstallation. These goals reflect a fundamental challenge of protecting users from unwanted software: it takes only one deceptive party in a chain of web advertisements, publishers, affiliate networks, and advertisers for abuse to manifest. It remains to be seen whether the approach taken by the Clean Software Alliance will yield the right balance between software monetization and user advocacy.

Acknowledgments

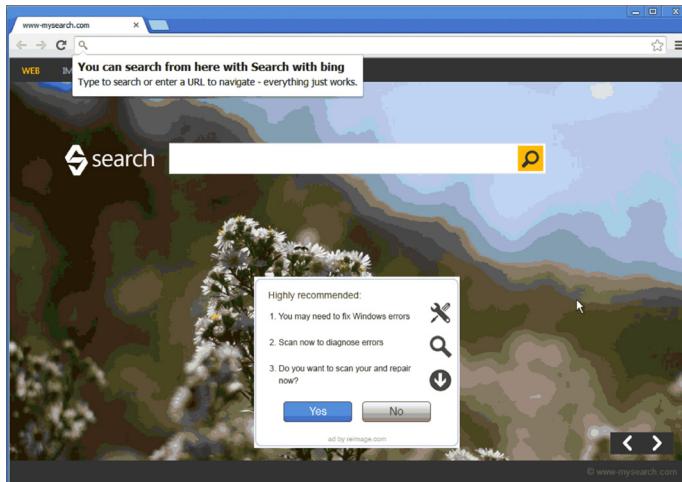
We thank the Safe Browsing and Chrome Security team for their insightful feedback in the development of our study on unwanted software and pay-per-install. This work was supported in part by the National Science Foundation under grants 1619620 and by a gift from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

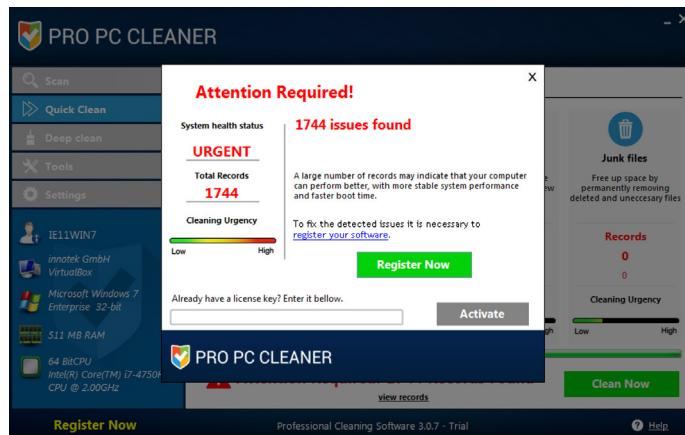
- [1] AVG. Become an AVG affiliate. <http://www.avg.com/affiliate/us-en/become-an-avg-affiliate>, 2016.
- [2] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Conference*, 2009.
- [3] Business Wire. Perion partners with lenovo to create lenovo browser guard. <http://www.businesswire.com/news/home/20140618005930/en/Perion-Partners-Lenovo-Create-Lenovo-Browser-Guard>, 2014.
- [4] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of the USENIX Security Symposium*, 2011.
- [5] Chrome. Chrome cleanup tool. <https://www.google.com/chrome/cleanup-tool/>, 2016.
- [6] Clean Software Alliance. Sustainable, consumer-friendly practices. <http://www.cs-alliance.org/>, 2016.
- [7] Comodo. Consumer affiliate. <https://www.comodo.com/partners/consumer-affiliate.php>, 2016.
- [8] CrunchBase. InstallMonetizer. <https://www.crunchbase.com/organization/installmonetizer#/entity>, 2016.
- [9] CrunchBase. OpenCandy. <https://www.crunchbase.com/product/opencandy#/entity>, 2016.
- [10] Nishant Doshi, Ashwin Athalye, and Eric Chien. Pay-Per-Install The New Malware Distribution Network. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/pay_per_install.pdf, 2010.
- [11] Ben Edelman. Claria’s misleading installation methods - ezone.com. <http://www.benedelman.org/spyware/installations/ezone-claria/>, 2005.
- [12] Ben Edelman. Pushing spyware through search. <http://www.benedelman.org/news/012606-1.html>, 2006.
- [13] gensim. models.ldamodel – Latent Dirichlet Allocation. <https://radimrehurek.com/gensim/models/ldamodel.html>, 2015.
- [14] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the Conference on Computer and Communications Security*, 2012.
- [15] Orr Hirschauge. Conduit diversifies away from ‘download valley’. <http://www.wsj.com/articles/SB10001424052702304579563281761548844>, 2014.
- [16] HowToGeek. Here’s what happens when you install the top 10 download.com apps. <http://www.howtogeek.com/198622/heres-what-happens-when-you-install-the-top-10-download.com-apps/>, 2014.
- [17] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and lessons from three years fighting malicious extensions. In *Proceedings of the USENIX Security Symposium*, 2015.
- [18] Java. What are the ask toolbars? https://www.java.com/en/download/faq/ask_toolbar.xml, 2015.
- [19] Markus Kammerstetter, Christian Platzer, and Gilbert Wondracek. Vanity, cracks and malware: Insights into the anti-copy protection ecosystem. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.
- [20] Platon Kotzias, Leyla Bilge, and Juan Caballero. Measuring PUP Prevalence and PUP Distribution through Pay-Per-Install Services. In *Proceedings of the USENIX Security Symposium*, 2016.
- [21] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. Certified PUP: Abuse in Authenticode Code Signing. In *Proceedings of the 22nd ACM Conference on Computer and Communication Security*, 2015.
- [22] Christian Kreibich, Nicholas Weaver, Chris Kanich, Weidong Cui, and Vern Paxson. Gq: Practical containment for measuring modern malware systems. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, 2011.
- [23] Bum Jun Kwon, Jayanta Mondal, Jiyoung Jang, Leyla Bilge, and Tudor Dumitras. The Dropper Effect: Insights into Malware Distribution with Downloader Graph Analytics. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS ’15*, pages 1118–1129, 2015.

- [24] LavaSoft. LavaSoft affiliate program. <http://affiliates.lavasoft.com/>, 2016.
- [25] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A crawler-based study of spyware in the web. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*, 2006.
- [26] Terry Nelms, Roberto Perdisci, Manos Antonakakis, and Mustaque Ahmad. Towards Measuring and Mitigating Social Engineering Malware Download Attacks. In *Proceedings of the USENIX Security Symposium*, 2016.
- [27] Niels Provos. All about safe browsing. <http://blog.chromium.org/2012/01/all-about-safe-browsing.html>, 2012.
- [28] Moheeb Abu Rajab. Year one: progress in the fight against unwanted software. <https://googleonlinesecurity.blogspot.com/2015/12/year-one-progress-in-fight-against.html>, 2015.
- [29] Moheeb Abu Rajab, Lucas Ballard, Noé Lutz, Panayiotis Mavrommatis, and Niels Provos. Camp: Content-agnostic malware protection. In *Proceedings of the Network and Distributed System Security Conference*, 2013.
- [30] Christian Rossow, Christian Dietrich, and Herbert Bos. Large-scale analysis of malware downloaders. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, pages 42–61, 2013.
- [31] Ben Fox Rubin. Perion sees soaring 2014 earnings following merger. <http://www.wsj.com/news/articles/SB10001424052702304815004579417252707242262>, 2014.
- [32] Stefan Saroiu, Steven D. Gribble, and Henry M. Levy. Measurement and analysis of spyware in a university environment. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 11–11, 2004.
- [33] Kevin Stevens. The Underground Economy of the Pay-Per-Install (PPI) Business. <http://www.secureworks.com/cyber-threat-intelligence/threats/ppi/>, 2009.
- [34] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.
- [35] TrendMicro. On the Actors Behind MEVADE/SEFNIT. <http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-on-the-actors-behind-mevade-sefnit.pdf>, 2014.
- [36] VirusTotal. VirusTotal. <https://www.virustotal.com/>, 2016.
- [37] China Internet Watch. Qihoo 360 launched its own affiliate network. <http://www.chinainternetwatch.com/7960/qihoo-360-launched-its-own-affiliate-network/>, 2014.
- [38] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

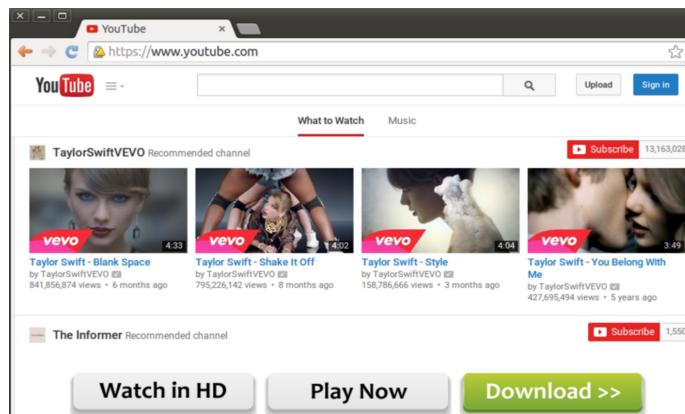
Appendix



(a) Browsing settings hijacker that overrides a victim's default search, supplying the traffic to Bing. The search page also displays ads for more unwanted software.

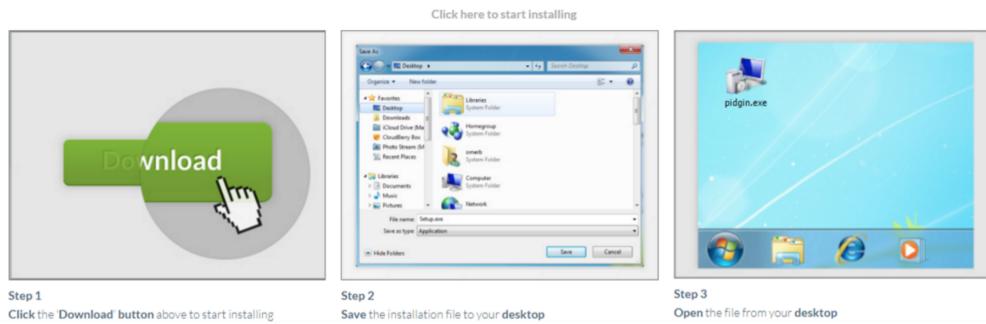


(b) Scareware that scans a victim's machine and reports thousands of urgent system health issues. Fixing these requires that victims pay a subscription fee.



(c) Ad injector that inserts advertisements into pages a victim visits. In this case, the ads direct to more unwanted software.

Sample of user experiences for the software bundled via pay-per-install.



- (a) PPI networks previously instructed victims to download applications via a Flash dialogue in order to abuse a bug in Chrome that prevented Safe Browsing from inspecting the downloaded file.



- (b) PPI network previously instructed victims to download password-protected compressed executables in order to prevent inspection of the downloaded file by Safe Browsing.

Sample of now defunct techniques employed by PPI networks to deliver PPI downloaders while evading Safe Browsing.

Measuring PUP Prevalence and PUP Distribution through Pay-Per-Install Services

Platon Kotzias

IMDEA Software Institute &
Universidad Politécnica de
Madrid, Spain
platon.kotzias@imdea.org

Leyla Bilge

Symantec Research Labs
Sofia Antipolis, France
leyla_bilge@symantec.com

Juan Caballero

IMDEA Software Institute
Madrid, Spain
juan.caballero@imdea.org

Abstract

Potentially unwanted programs (PUP) such as adware and rogueware, while not outright malicious, exhibit intrusive behavior that generates user complaints and makes security vendors flag them as undesirable. PUP has been little studied in the research literature despite recent indications that its prevalence may have surpassed that of malware.

In this work we perform the first systematic study of PUP prevalence and its distribution through pay-per-install (PPI) services, which link advertisers that want to promote their programs with affiliate publishers willing to bundle their programs with offers for other software. Using AV telemetry information comprising of 8 billion events on 3.9 million real hosts during a 19 month period, we discover that over half (54%) of the examined hosts have PUP installed. PUP publishers are highly popular, e.g., the top two PUP publishers rank 15 and 24 amongst all software publishers (benign and PUP). Furthermore, we analyze the who-installs-who relationships, finding that 65% of PUP downloads are performed by other PUP and that 24 PPI services distribute over a quarter of all PUP. We also examine the top advertiser programs distributed by the PPI services, observing that they are dominated by adware running in the browser (e.g., toolbars, extensions) and rogueware. Finally, we investigate the PUP-malware relationships in the form of malware installations by PUP and PUP installations by malware. We conclude that while such events exist, PUP distribution is largely disjoint from malware distribution.

1 Introduction

Potentially unwanted programs (PUP) are a category of undesirable software that includes adware and rogue

software (i.e., rogueware). While not outright malicious (i.e., malware), PUP behaviors include intrusive advertising such as ad-injection, ad-replacement, pop-ups, and popunders; bundling programs users want with undesirable programs; tracking users' Internet surfing; and pushing the user to buy licenses for rogueware of dubious value, e.g., registry optimizers. Such undesirable behaviors prompt user complaints and have led security vendors to flag PUP in ways similar to malware.

There exist indications that PUP prominence has quickly increased over the last years. Already in Q2 2014, AV vendors started alerting of a substantial increase in collected PUP samples [59]. Recently, Thomas et al. [64] showed that ad-injectors, a popular type of PUP that injects advertisements into user's Web surfing, affects 5% of unique daily IP addresses accessing Google [64]. And, Kotzias et al. [35] measured PUP steadily increasing since 2010 in (so-called) malware feeds, to the point where nowadays PUP samples outnumber malware samples in those feeds. Still, the prevalence of PUP remains unknown.

A fundamental difference between malware and PUP is distribution. Malware distribution is dominated by *silent* installation vectors such as drive-by downloads [22, 53], where malware is dropped through vulnerability exploitation. Thus, the owner of the compromised host is unaware a malware installation happened. In contrast, PUP does not get installed silently because that would make it malware for most AV vendors. A property of PUP is that it is installed with the consent of the user, who (consciously or not) approves the PUP installation on its host.

In this work, we perform the first systematic study of PUP prevalence and its distribution through pay-per-install (PPI) services. PPI services (also called PPI net-

works) connect advertisers willing to buy installs of their programs with affiliate publishers selling installs. The PPI services used for distributing PUP are disjoint from silent PPI services studied by prior work [7]. Silent PPI services are exclusively used for malware distribution, while the PPI services we study are majoritarily used for distributing PUP and benign software. In the analyzed PPI services, an affiliate publisher owns an original program (typically freeware) that users want to install. To monetize installations of its free program, the affiliate publisher bundles (or replaces) it with an installer from a PPI service, which it distributes to users looking for the original program. During the installation process of the original program, users are prompted with offers to also install other software, belonging to advertisers that pay the PPI service for successful installs of their advertised programs.

To measure PUP prevalence and its distribution through PPI services we use AV telemetry information comprising 8 billion events on 3.9 million hosts during a 19 month time period. This telemetry contains events where parent programs installed child programs and we focus on events where the publishers of either parent or child programs are PUP publishers. This data enables us to measure the prevalence of PUP on real hosts and to map the who-installs-who relationships between PUP publishers, providing us with a broad view of the PUP ecosystem.

We first measure PUP prevalence by measuring the installation base of PUP publishers. We find that programs from PUP publishers are installed in 54% of the 3.9M hosts examined. That is, more than half the examined hosts have PUP. We rank the top PUP publishers by installation base and compare them with benign publishers. The top two PUP publishers, both of them PPI services, are ranked 15 and 24 amongst all software publishers (benign or not). The top PUP publisher is more popular than NVIDIA, a leading graphics hardware manufacturer. The programs of those two top PUP publishers are installed in 1M and 533K hosts in our AV telemetry dataset, which we estimate to be two orders of magnitude higher when considering all Internet-connected hosts. We estimate that each top 20 PUP publisher is installed on 10M–100M hosts.

We analyze the who-installs-who relationships in the publisher graph to identify and rank top publishers playing specific roles in the ecosystem. This enables us to identify 24 PPI services distributing PUP in our analyzed time period. We also observe that the top PUP advertisers predominantly distribute browser add-ons involved

in different types of advertising and by selling software licenses for rogueware. We measure PUP distribution finding that 65% of PUP downloads are performed by other PUP, that the 24 identified PPI services are responsible for over 25% of all PUP downloads, and that advertiser affiliate programs are responsible for an additional 19% PUP downloads.

We also examine the malware-PUP relationships, in particular how often malware downloads PUP and PUP downloads malware. We find 11K events (0.03%) where popular malware families install PUP for monetization and 5,586 events where PUP distributes malware. While there exist cases of PUP publishers installing malware, PUP–malware interactions are not prevalent. Overall, it seems that PUP distribution is largely disjoint from malware distribution. Finally, we analyze the top domains distributing PUP, finding that domains from PPI services dominate by number of downloads.

Contributions:

- We perform the first systematic study of PUP prevalence and its distribution through PPI services using AV telemetry comprising 8B events on 3.9M hosts over a 19-month period.
- We measure PUP prevalence on real hosts finding that 54% have PUP installed. We rank the top PUP publishers by installation base, finding that the top two PUP publishers rank 15 and 24 amongst all (benign and PUP) software publishers. We estimate that the top 20 PUP publishers are each installed on 10M–100M hosts.
- We build a publisher graph that captures the who-installs-who relationships between PUP publishers. Using the graph we identify 24 PPI services and measure that they distribute over 25% of the PUP.
- We examine other aspects of PUP distribution including downloads by advertiser affiliate programs, downloads of malware by PUP, downloads of PUP by malware, and the domains from where PUP is downloaded. We conclude that PUP distribution is largely disjoint from malware distribution.

2 Overview and Problem Statement

This section first introduces the PPI ecosystem (Section 2.1), then details the datasets used (Section 2.2), and finally describes our problem and approach (Section 2.3).

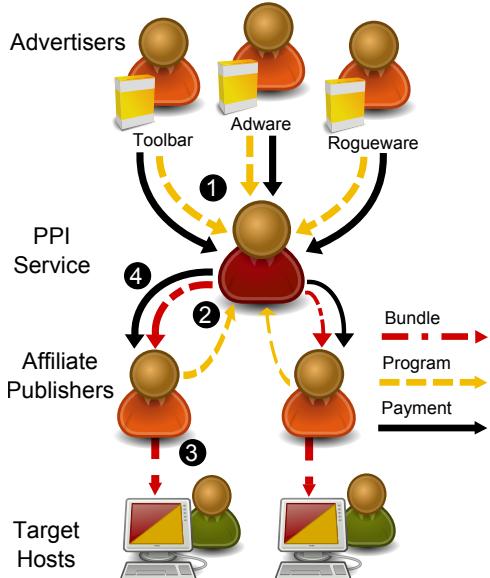


Figure 1: Typical transactions in the PPI market. (❶) Advertisers provide software they want to have installed, and pay a PPI service to distribute it. (❷) Affiliate publishers register with the PPI service, provide their program, and receive a bundle of their program with the PPI installer. (❸) Affiliate publishers distribute their bundle to target users. (❹) The PPI service pays affiliate publishers a bounty for any successful installations they facilitated.

2.1 Pay-Per-Install Overview

The PPI market, as depicted in Figure 1, consists of three main actors: *advertisers*, *PPI services/networks*, and *affiliate publishers*. *Advertisers* are entities that want to install their programs onto a number of target hosts. They wish to *buy installs* of their programs. The PPI service receives money from advertisers for the service of installing their programs onto the target hosts. They are called advertisers because they are willing to pay to promote their programs, which are offered to a large number of users by the PPI service. Advertiser programs can be benign, potentially unwanted (PUP), and occasionally malware.

Affiliate publishers are entities that *sell installs* to PPI services. They are often software publishers that own programs (e.g., freeware) that users may want to install, and who offer the advertiser programs to those users installing their programs. This enables affiliate publishers to monetize their freeware, or to generate additional income on top of the normal business model of their programs. They can also be website owners that offer visi-

Country	Avg	Range
United States	\$1.30	\$0.70-\$2.00
United Kingdom	\$0.80	\$0.40-\$1.50
Australia	\$0.40	\$0.30-\$0.50
Canada	\$0.40	\$0.30-\$0.50
France	\$0.28	\$0.15-\$0.50
Germany	\$0.25	\$0.10-\$0.40
New Zealand	\$0.23	\$0.15-\$0.35
Ireland	\$0.19	\$0.15-\$0.25
Denmark	\$0.18	\$0.15-\$0.20
Austria	\$0.16	\$0.15-\$0.20
Netherlands	\$0.16	\$0.10-\$0.20
Finland	\$0.15	\$0.10-\$0.20
Norway	\$0.15	\$0.05-\$0.20
Switzerland	\$0.12	\$0.03-\$0.20
Spain	\$0.11	\$0.03-\$0.20

Table 1: Top 15 countries with the highest average price per install collected from 3 PPI services [8, 9, 50] on June 2016.

tors to download an installer from the PPI service, thus selling installs on the visitor’s machines. Affiliate publishers are often referred simply as publishers, but we use publishers to refer to software owners, and affiliate publishers for those signing up to PPI services.

The PPI service acts as a middle man that buys installs from affiliate publishers and sells installs to advertisers. The PPI service credits the affiliate publisher a *bounty* for each confirmed installation, i.e., affiliate displays an offer for an advertised program *and* the user approves and successfully installs the advertised program.

Affiliate publishers are paid between \$2.00 and \$0.01 per install depending on the geographic location. Prices vary over time based on offer and demand and the current price is typically only available to registered affiliate publishers. Table 1 shows the prices paid to affiliate publishers for the most demanded countries on June 25th, 2016 by 3 PPI services that publicly list their prices to attract affiliate publishers. The highest paid country is the United States with an average install price of \$1.30, followed by the United Kingdom (\$0.80), Australia and Canada (\$0.40), and European countries starting at \$0.30 for France. The cheapest installs are \$0.03–\$0.01 for Asian and African countries (typically part of a “Rest of the World” region). In comparison, prices paid to affiliate publishers by silent PPI services that distribute malware range \$0.18–\$0.01 per install [7]. This shows that malware distribution can be an order of magnitude cheaper for the most demanded countries.

A common PPI model (depicted in Figure 1) is that the affiliate publisher provides the PPI service with its program executable and the PPI service *wraps* (i.e., bundles) the affiliate’s program with some *PPI installer* software, and returns the bundle/wrapper to the affiliate publisher. The affiliate publisher is then in charge of distributing the bundle to users interested in the affiliate’s program. The distribution can happen through different vectors such as websites that belong to the affiliate publisher or uploading the bundle to *download portals* such as Download.com [15] or Softonic [61]. When a user executes the wrapper, the wrapper installs the affiliate’s program and during this installation it offers the user to install other advertised programs. If the user downloads and installs one of the offers, the PPI service pays a bounty to the affiliate’s account.

An affiliate publisher can register with a PPI service even if it does not own programs that users want to install. Some PPI services look for affiliate website owners whose goal is to convince visitors of their websites to download and run an installer from the PPI service. Furthermore, some PPI services offer a *pre-wrapped software* model where the PPI service wraps its own software titles with the advertiser offers, and provides the bundle to the affiliate publishers [29]. Some PPI services even allow affiliate publishers to monetize on third-party free programs (e.g., GNU).

Some download portals such as Download.com run their own PPI service. When publishers upload their programs to the portal (e.g., through Upload.com) they are offered if they want to monetize their programs. If so, the download portal wraps the original program and makes the bundle available for download. In this model the download portal is in charge of distribution.

Another distribution model are *affiliate programs* where an advertiser uses affiliate publishers to distribute its software directly, without a PPI service. This is a one-to-many distribution model, in contrast with the many-to-many distribution model of PPI services.

2.2 Datasets

Our paper leverages several datasets to conduct a systematic investigation about PUP prevalence and distribution. We analyze WINE’s binary downloads dataset [16] to trace PUP installations by real users and their parent/child (downloader/downloadee) relationships, the list of signed malicious executables from the Malsign project [35] to cluster together executables signed by different signers that belong to the same publisher,

Dataset	Data	Count
WINE	Events Analyzed	8 B
01/2013 – 07/2014	Events with Parent	90 M
	Total number of Machines	3.9 M
	All Files	2.6 M
	Parent Files	657 K
	Child Files	2 M
	Signed Files	982 K
	Publishers	6 K
	Parent Publishers	1.4 K
	Child Publishers	6 K
	Events with URL	1.1 M
	URLs	290 K
	FQDNs	13.4 K
	ESLDs	7.5 K
Malsign	Signed executables	142 K
VirusTotal	Reports	12 M
	Feed Reports	11 M
	WINE Reports	1.1 M
	Malsign Reports	142 K

Table 2: Summary of datasets used.

and VirusTotal [67] reports for enriching the previous datasets with additional file meta-data (e.g., AV detections, file certificates for samples not in Malsign). Table 2 summarizes these datasets.

WINE. The Worldwide Intelligence Network Environment (WINE) [17] provides researchers a platform to analyze data collected from Symantec customers that opt-in to the collection. This data consists of anonymous telemetry reports about security events (e.g., AV detections, file downloads) on millions of real computers in active use around the world.

In this work, we focus on the *binary downloads* dataset in WINE, which records meta-data about all executable files (e.g., EXE, DLL) and compressed archives (e.g., ZIP, RAR, CAB) downloaded by Windows hosts regardless if they are malicious or benign. Each *event* in the dataset can correspond to (1) a download of an executable file or compressed archive over the network, or (2) the extraction of a file from a compressed archive. For our work, we analyze the following fields: the server-side timestamp of the event, the unique identifier for the machine where the event happens, the SHA256 hash of the child file (i.e., downloaded or extracted), the SHA256 hash of the parent process (i.e., downloader program or decompressing tool), the certificate subject for the parent and child files if they are signed, and, when available, the URL from where the child file was downloaded. The files themselves are not included in the dataset.

We focus our analysis on the 19 months between January 1st 2013 and July 23rd 2014. As our goal is to analyze PUP (i.e., executables from PUP publishers), we only monitor the downloads of PUP and the files that are downloaded by PUP, i.e., events where either the child or the parent is PUP. This data corresponds to 8 billion events. The details of the data selection methodology are explained in Section 3. Out of 8 B events, 90 M events have information about the parent file that installed the PUP. Those events comprise 2.6 M distinct executables out of which 982 K (38%) are signed by 6 K publishers.

A subset of 1.1 M events provide information about the URL the child executable was downloaded from. These events contain 290 K unique URLs from 13.4 K fully qualified domain names (FQDNs). To aggregate the downloads initiated from the same domain owner, we extract the effective second-level domain (ESLD) from the FQDN. For example, the ESLD of `www.google.com` is the 2LD `google.com`, however, the ESLD of `www.amazon.co.uk` is the 3LD `amazon.co.uk` since different entities can request `co.uk` subdomains. We extract the ESLDs of the domains by consulting Mozilla’s public suffix list [54].

Malsign. To cluster executables in the WINE binary downloads dataset signed by different entities that belong to the same publisher, we leverage a dataset of 142 K signed malware and PUP from the Malsign project [35]. This dataset includes the samples and their clustering into families. The clustering results are based on statically extracted features from the samples with a focus on features from the Windows Authenticode signature [39]. These features include: the leaf certificate hash, leaf certificate fields (i.e., public key, subject common name and location), the executable’s hash in the signature (i.e., Authentihash), file metadata (i.e., publisher, description, internal name, original name, product name, copyright, and trademarks), and the PEhash [68]. From the clustering results we extract the list of publisher names (subject common name in the certificates) in the same cluster, which should belong to the same publisher.

VirusTotal. VirusTotal [67] is an online service that analyzes files and URLs submitted by users. One of its services is to scan the submitted binaries with anti-virus products. VirusTotal also offers a web API to query meta-data on the collected files including the AV detection rate and information extracted statically from the files. We use VirusTotal to obtain additional meta-data about the WINE files, as well as from 11 M malicious/undesirable executables from a feed. In particular, we obtain: AV detection labels for the sample, first seen

timestamp, detailed certificate information, and values of fields in the PE header. This information is not available otherwise as we do not have access to the WINE files that are not in Malsign, but we can query VirusTotal using the file hash. We consider that a file is malicious if at least 4 AV engines in the VT report had a detection label for it, a threshold also used in prior works to avoid false positives [35].

2.3 Problem Statement

In this paper we conduct a systematic analysis of PUP prevalence and its distribution through PPI services. We split our measurements in two main parts. First, we measure how prevalent PUP is. This includes what fraction of hosts have PUP installed, which are the top PUP publishers, and what is the installation base of PUP publishers in comparison with benign publishers. Then, we measure the PPI ecosystem including who are the top PPI services and PUP advertisers, what percentage of PUP installations are due to PPI services and advertiser affiliate programs, what are the relationships between PUP and malware, and what are the domains from where PUP is downloaded.

We do not attempt to differentiate what behaviors make a program PUP or malware, but instead rely on AV vendors for this. We leverage the prior finding that the majority of PUP (and very little malware) is properly signed. In particular, signed executables flagged by AV engines are predominantly PUP, while malware rarely obtains a valid code signing chain due to identity checks implemented by CAs [35]. Using that finding, we consider PUP any signed file flagged by at least 4 AV engines. Thus, the term PUP in this paper includes different types of files that AV vendors flag as PUP including undesirable advertiser programs, bundles of publisher programs with PPI installers, and stand-alone PPI installers.

To measure PUP prevalence, we first identify a list of dominant PUP publishers extracted from the code signing certificates from the 11M VT reports from the malware feed (Section 3). Then, we group publisher names (i.e., subject strings in code signing certificates) from the same entity into publisher clusters (Section 4). Finally, we use the WINE binary reputation data to measure the PUP installation base, as well as the installation base of individual PUP and benign publisher clusters (Section 5). Since we focus on signed executables, our numbers constitute a lower bound on PUP and publisher prevalence.

To measure the PPI ecosystem, we build a publisher graph that captures the who-installs-who relationships

among PUP publishers. We use the graph for identifying PPI services and PUP advertisers (Section 6). Then, we measure the percentage of PUP installations due to PPI services and advertiser affiliate programs (Section 7). Next, we analyze the downloads of malware by PUP and the downloads of PUP by malware (Section 8). Finally, we examine the domains from where PUP is downloaded (Section 9).

3 Identifying PUP Publishers

The first step in our approach is to identify a list of dominant PUP publishers. As mentioned earlier, prior work has shown that signed executables flagged by AV engines are predominantly PUP, while malware is rarely properly signed. Motivated by this finding, we identify PUP publishers by ranking publishers of signed binaries flagged by AV vendors, by the number of samples they sign.

For this, we obtain a list of 11M potentially malicious samples from a “malware” feed and query them in VirusTotal to collect their VT reports. From these reports, we keep only executables flagged by at least 4 AV vendors to make sure we do not include benign samples in our study. We further filter out executables with invalid signatures, i.e., whose publisher information cannot be trusted. These filtering steps leave us with 2.5M binaries whose signatures validated at the time of signing. These include executables whose certificate chain still validates, those with a revoked certificate, and those with expired certificates issued by a valid CA.

From each of the 2.5M signed executables left, we extract the publisher’s subject common name from the certificate information in its VT report. Hereinafter, we will refer to the publisher’s subject common name as publisher name. Oftentimes, publisher names have some variations despite belonging to the same entity. For example, MyRealSoftware could use both “MyRealSoftware S.R.U” and “MyRealSoftware Inc” in the publisher name. Thus, we perform a normalization on the publisher names to remove company suffixes such as Inc., Ltd. This process outputs a list of 1,440 normalized PUP publisher names. Table 11 in the Appendix shows the top 20 normalized PUP publisher names by number of samples signed in the feed. These 20 publishers own 56% of the remaining signed samples after filtering.

Clearly, our list does not cover all PUP publishers in the wild. This would not be possible unless we analyzed all existing signed PUP. However, the fact that we analyze 2.5M of undesirable/malicious signed samples gives us confidence that we cover the top PUP publishers.

Those 1,440 PUP publisher names are used to scan the file publisher field in WINE’s binary downloads dataset to identify events that involve samples from those PUP publishers, i.e., where a parent or child file belongs to the 1,440 PUP publishers. As shown in Table 2, there are 8 B such events.

Note that at this point we still do not know whether different publisher names (i.e., entries in Table 11) belong to the same PUP publisher. For example, some popular publisher names such as Daniel Hareuveni, Stepan Rybin, and Stanislav Kabin are all part of Web Pick Internet Holdings Ltd, which runs the InstalleRex PPI service. The process to cluster publisher names that belong to the same publisher is described in Section 4.

4 Clustering Publishers

PUP authors use certificate polymorphism to evade detection by certification authorities and AV vendors [35]. Two common ways to introduce certificate polymorphism are applying small variations to reuse the same identity / publisher name (e.g. apps market ltd, APPS Market Inc., Apps market Incorporated) and using multiple identities (i.e., companies or persons) to obtain code signing certificates. We cluster publisher names that belong to the same publisher according to similarities on the publisher names, domain names in events with URLs, and Malsign clustering results.

Publisher name similarity. This feature aims to group together certificates used by the same identity that have small variations on the publisher name. Since the WINE binary downloads dataset contains the publisher name for parent and child files, this feature can be used even when a signed sample has no VT report and we do not have the executable (i.e., not in Malsign). The similarity between two publisher names is computed in two parts: first derive a list of normalized tokens from each publisher name through four steps and then compute similarity between the token lists.

To obtain the token list of a publisher name, the first step is to extract parenthesized strings as separate tokens. For example, given the publisher name “Start Playing (Start Playing (KnockApps Limited))” this step produces 3 tokens: “Start Playing”, “Start Playing”, and “KnockApps Limited”. The second step converts each token to lowercase and removes all non-alphanumeric characters from the token. The third step removes from the tokens company extensions (e.g., ltd, limited, inc, corp), geographical locations (e.g., countries, cities), and the string

Publishers	Clusters	Singletons	Largest	Median
6,066	5,074	4,534	103	1

Table 3: Publisher clustering results.

“Open Source Developer”, which appears in code signing certificates issued to individual developers of open source projects. Finally, tokens that have less than 3 characters and duplicate tokens are removed.

To compute the similarity between two token lists, for each pair of publisher names P_1 and P_2 , we calculate the normalized edit distance among all token pairs (t_i, t_j) where t_i belongs to P_1 and t_j to P_2 . If the edit distance between P_1 and P_2 is less than 0.1, we consider these two publishers to be the same. We selected this threshold after experimenting with different threshold values over 1,157 manually labeled publisher names. The edit distance threshold of 0.1 allowed us grouping the 1,157 publisher names into 216 clusters with 100% precision, 81.9% recall, and 86.4% F1 score.

Child download domains. If child executables signed by different publisher names are often downloaded from the same domains, that is a good indication that the publisher names belong to the same entity. To capture this behavior, we compute the set of ESLDs from where files signed by the same publisher name have been downloaded. Note that we exclude ESLDs that correspond to file lockers and download portals as they are typically used by many different publishers. The publisher names whose Jaccard Index of their ESLD sets is over 0.5 are put to the same cluster.

Parent download domains. Similarly, if parent files signed by different publisher names download from a similar set of domains, this indicates the publisher names likely belong to the same entity. This feature first computes the set of ESLDs from where parent files signed by the same publisher name download (excluding file lockers and download portals). Publisher names whose Jaccard Index is over 0.5 are put to the same cluster.

Malsign clustering. For each Malsign cluster we extract the list of distinct publisher names used to sign executables in the cluster, i.e., Subject CN strings extracted from certificates for files in the cluster. We consider that two publisher names in the same Malsign cluster belong to the same publisher.

Final clustering. We group publisher names into the same cluster if they satisfy at least one of the first 3 features explained above or are in the same Malsign cluster. Table 3 summarizes the clustering, which produces 5,074 clusters from 6,066 publisher names.

5 PUP Prevalence

In this Section, we measure the prevalence of PUP, based on the number of hosts in the WINE binary downloads dataset (i.e., WINE hosts) that have installed programs from PUP publishers. We measure the total number of WINE hosts affected by PUP, rank PUP publishers by installation base, and compare the installation base of PUP publishers to benign publishers.

We first compute the *detection ratio* (DR) for each cluster, which is the number of samples signed by publishers in the cluster flagged by at least 4 AVs, divided by the total number of samples in the cluster for which we have a VT report. We mark as PUP those clusters with $DR > 5\%$, a threshold chosen because is the lowest that leaves out known benign publishers. From this point on, when we refer to PUP publishers, we mean the 915 publisher clusters with $DR > 5\%$.

Note that the number of WINE hosts with installed programs from a publisher cluster constitutes a quite conservative lower bound on the number of hosts across the Internet that have programs installed from that publisher. It captures only those Symantec customers that have opted-in to share data and have been sampled into WINE. If we take into account that Symantec only had 8% of the AV market share in January 2014 [47] and that only $\frac{1}{16}$ of Symantec users that opt-in to share telemetry are sampled into WINE [6], we estimate that the number of WINE hosts is two orders of magnitude lower than the corresponding number of Internet-connected hosts. Furthermore, we do not count WINE hosts with only unsigned PUP executables installed.

PUP prevalence. We find 2.1M WINE hosts, out of a total 3.9M WINE hosts in our time period, with at least one executable installed from the 915 PUP clusters. Thus, 54% of WINE hosts have PUP installed. This ratio is a lower bound because we only count signed PUP executables (i.e., we ignore unsigned PUP executables) and also because our initial PUP publisher list in Section 3 may not be complete. Thus, PUP is prevalent: more than half of the hosts examined have some PUP installed.

Top PUP publishers. Table 4 shows the top 20 PUP publishers by WINE installation base and details the cluster name, whether the publisher is a PPI service (this classification is detailed in Section 6), the number of publisher names in the cluster, detection ratio, and host installation base. The number of publishers ranks from singleton clusters up to 48 publishers for IronSource, an Israeli PPI service. The installation bases for the top 20 PUP publishers range from 200K up to over 1M for Pe-

#	Cluster	PPI	Pub	DR	Hosts
1	Perion Network	✓	5	52%	1.0M
2	Mindspark	✗	1	85%	533K
3	Bandoo Media	✗	5	46%	373K
4	Web Pick	✓	21	79%	346K
5	IronSource	✓	48	81%	332K
6	Babylon	✗	1	38%	330K
7	JDI BACKUP	✗	1	56%	328K
8	Systweak	✗	3	37%	320K
9	OpenCandy	✓	1	55%	311K
10	Montiera Technologies	✗	2	54%	303K
11	Softonic International	✗	2	70%	292K
12	PriceGong Software	✗	1	18%	292K
13	Adknowledge	✓	7	75%	277K
14	Adsology	✗	2	77%	276K
15	Visual Tools	✗	2	70%	275K
16	BitTorrent	✗	1	40%	271K
17	Wajam	✗	2	87%	218K
18	W3i	✓	4	93%	216K
19	iBario	✓	15	84%	208K
20	Tuguu	✓	14	94%	200K

Table 4: Top 20 PUP publishers by installation base.

riion Network, an Israeli PPI service that bought the operations of the infamous Conduit toolbar in 2013. As explained earlier, these numbers are a quite conservative lower bound. We estimate the number of Internet-connected computers for these publishers to be two orders of magnitude larger, in the range of tens of millions, and up to a hundred million, hosts. We have found anecdotal information that fits these estimates. For example, an adware developer interviewed in 2009 claimed to have control over 4M machines [12].

Comparison with benign publishers. Table 5 shows the top 20 publisher clusters, benign and PUP, in WINE. The most common publishers are Microsoft and Symantec that are installed in nearly all hosts. The Perion Network / Conduit PPI network ranks 15 overall. That is, there are only 14 benign software publishers with a larger installation base than the top PUP publisher. Perion Network is more prevalent than well known publishers such as Macrovision and NVIDIA. The second PUP publisher (Mindspark Interactive Network) has the rank 24. This highlights that top PUP publishers are among the most widely installed software publishers.

A reader may wonder if we could also compute the installation base for malware families. Unfortunately, due to malware being largely unsigned and highly polymorphic, we would need to first classify millions of files in WINE (without having access to the binaries) before we can perform the ranking.

#	Cluster	PUP	Hosts
1	Microsoft	✗	3.9M
2	Symantec	✗	3.8M
3	Adobe Systems	✗	3.5M
4	Google	✗	3.1M
5	Apple	✗	1.8M
6	Intel	✗	1.6M
7	Sun Microsystems	✗	1.6M
8	Cyberlink	✗	1.6M
9	GEAR Software	✗	1.5M
10	Hewlett-Packard	✗	1.5M
11	Oracle	✗	1.4M
12	Skype Technologies	✗	1.3M
13	Mozilla Corporation	✗	1.0M
14	McAfee	✗	1.0M
15	Perion Network / Conduit	✓	1.0M
16	WildTangent	✗	941K
17	Macrovision Corporation	✗	802K
18	LEAD Technologies	✗	775K
19	NVIDIA Corporation	✗	722K
20	Ask.com	✗	624K
24	Mindspark Interactive Network	✓	533K

Table 5: Top publishers by install base (benign and PUP).

6 Classifying Publishers

Among the 5,074 PUP publisher clusters obtained in Section 4 we want to identify important clusters playing a specific role in the ecosystem. In particular, we want to identify clusters that correspond to PPI services and to examine the type of programs distributed by the dominant advertisers. For this, we first build a *publisher graph* that captures the who-installs-who relationships. Then, we apply filtering heuristics on the publisher graph to select a subset of publishers that likely hold a specific role, e.g., PPI service. Finally, we manually classify the filtered publishers into roles by examining Internet resources, e.g., publisher web pages, PPI forums, and the Internet Archive [32].

Publisher graph. The publisher graph is a directed graph where each publisher cluster is a node and an edge from cluster C_A to cluster C_B means there is at least one event where a parent file from C_A installed a child file from C_B . Self-edges are excluded, as those indicate program updates and downloads of additional components from the same publisher. Note that an edge captures download events between parent and child clusters across all hosts and the 19 months analyzed. Thus, the publisher graph captures the who-installs-who relationships over that time period, enabling a birds-eye view of the ecosystem.

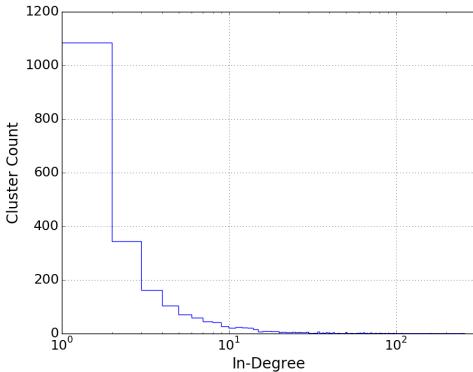
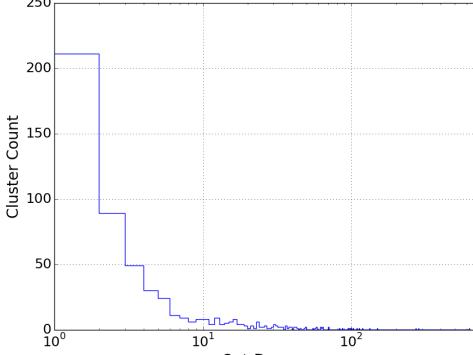


Figure 2: Cluster in-degree distribution.



#	Cluster	PPI Service	ID	OD	Hosts	DR	Pub.
1	Perion Network/Conduit	CodeFuel [10]	168	63	1 M	52%	5
2	Yontoo	Sterkly [63]	53	17	601 K	93%	103
3	iBario	RevenueHits [56]	62	36	479 K	84%	16
4	Web Pick	InstalleRex [27]	65	22	346 K	79%	21
5	IronSource	InstallCore [26]	73	112	332 K	81%	48
6	OpenCandy	OpenCandy [46]	91	36	311 K	55%	1
7	Adknowledge	Adknowledge [20]	53	48	277 K	75%	7
8	W3i	NativeX [41]	38	49	216 K	93%	4
9	Somoto	BetterInstaller [5]	60	70	209 K	96%	5
10	Firseria	Solimba [62]	41	30	209 K	94%	9
11	Tuguu	DomaIQ [13]	49	16	200 K	94%	14
12	Download Admin	DownloadAdmin [14]	25	16	192 K	73%	2
13	Air Software	AirInstaller [3]	33	41	191 K	79%	1
14	Vittalia Internet	OneInstaller [45]	27	29	155 K	71%	18
15	Amonetize	installPath [31]	50	63	154 K	93%	2
16	SIEN	Installbay [25]	34	33	139 K	80%	2
17	OutBrowse	RevenYou [57]	22	41	86 K	94%	4
18	Verti Technology Group	Verti [66]	17	39	47 K	44%	1
19	Blisbury	Smart WebAds [60]	19	30	46 K	77%	2
20	Nosibay	Nosibay [44]	19	20	30 K	75%	1
21	ConversionAds	ConversionAds [11]	10	38	24 K	72%	1
22	Installer Technology	InstallerTech [28]	10	14	11 K	56%	1
23	7install	7install [1]	2	0	75	12%	1
24	Install Monster	Install Monster [30]	3	1	9	100%	1

Table 6: PPI services services identified sorted by installation base.

(or gained popularity) after the end of our observation period (e.g., AdGazelle). Third, some PPI services may distribute unsigned bundles or downloaders. For example, we examined over 30K samples that AV engines label as belonging to the InstallMonetizer PPI service, of which only 8% were signed. Finally, some PPI services may have so low volume that they were not observed in our initial 11 M sample feed.

Advertisers. To identify advertisers in the publisher graph, we first select PUP clusters with high in-degree, low out-degree, and for which at least one parent is one of the 24 PPI services (i.e., $DR \geq 5\% \wedge ID \geq 10 \wedge OD \leq 9 \wedge PPPI > 0$). Advertisers pay to have their products installed (i.e., buy installs) and may not install other publishers for monetization as they know how to monetize the machines themselves. Since buying installs costs money, they need to generate enough income from the installations to offset that cost. This filtering identifies 77 clusters, which we manually examine to identify the main product they advertise (they can advertise multiple ones) and whether they run an affiliate program where they pay affiliates to distribute their programs. We also include in this analysis the 20 advertiser clusters manually identified in the PPI service identification above.

Table 7 shows the top 30 advertiser clusters by installation base. The table shows the cluster name, whether it runs an affiliate program, in-degree, out-degree, detection ratio, installation base, the number of parent PPI service nodes, the number of child PPI service nodes, the main product they install, and whether they install browser add-ons (BAO). The latter includes any type of browser add-ons such as toolbars, extensions, plugins, browser helper objects, and sidebars.

The data shows that 18 of the 30 top advertisers install browser add-ons. Those browser add-ons enable monetization through Web traffic, predominantly through different types of advertisement. Common methods are modifying default search engines to monetize searches (e.g., SearchResults, Delta Toolbar, Imminent Toolbar), shopping deals and price comparisons (e.g., PriceGong, PricePeep, DealPLY, SupremeSavings), and other types of advertisement such as pay-per-impression and pay-per-action (e.g., Widgi Toolbar, Inbox Toolbar).

The 12 advertisers that focus on client applications monetize predominantly through selling licenses and subscriptions. The main group is 6 publishers advertising rogueware claiming to improve system performance (Regclean Pro, Optimizer Pro, SpeedUpMyPC,

#	Cluster	Aff	ID	OD	DR	Hosts	PPPI	CPPI	Main Product	BAO
1	Xacti	✗	57	9	22%	563 K	13	1	RebateInformer	✓
2	Mindspark	✓	62	17	85%	533 K	3	5	Mindspark Toolbar	✓
3	Bandoo Media	✓	86	108	46%	373 K	7	18	MediaBar	✓
4	Babylon	✓	83	14	38%	330 K	16	3	Babylon Toolbar	✓
5	JDI Backup Limited	✓	71	19	56%	328 K	17	3	MyPC Backup	✗
6	Systweak	✓	81	24	37%	320 K	7	2	Regclean Pro	✗
7	Montiera Technologies	✗	37	2	66%	303 K	8	1	Delta Toolbar	✓
8	PriceGong Software	✗	12	0	17%	292 K	6	0	PriceGong	✓
9	Adsology	✓	62	12	77%	276 K	17	1	OptimizerPro	✗
10	Wajam	✗	42	5	87%	218 K	11	2	Wajam	✓
11	Visicom Media	✗	13	2	14%	185 K	4	0	VMN Toolbar	✓
12	Linkury	✗	46	2	54%	174 K	13	0	SmartBar	✓
13	Uniblue Systems	✓	64	13	11%	160 K	10	1	SpeedUpMyPC	✗
14	Search Results	✗	35	3	79%	159 K	12	2	SearchResults	✓
15	Bitberry Software	✓	13	64	88%	130 K	1	7	BitZipper	✗
16	Iminent	✗	13	1	74%	118 K	4	1	Iminent Toolbar	✓
17	DealPly Technologies	✗	43	0	93%	108 K	16	0	DealPly	✓
18	Smart PC Solutions	✓	38	0	32%	106 K	13	0	PC Speed Maximizer	✗
19	DVDVideoSoft	✗	15	2	18%	101 K	3	1	Free Studio	✗
20	Spigot	✓	17	1	39%	101 K	1	1	Widgi Toolbar	✓
21	Web Cake	✗	34	2	98%	97 K	16	2	Desktop OS	✓
22	GreTech	✓	13	1	21%	90 K	3	1	GOM Player	✗
23	Digital River	✓	17	0	10%	80 K	1	0	DR Download Manager	✓
24	Widdit	✓	20	16	27%	79 K	4	2	HomeTab	✓
25	EpicPlay	✗	12	4	90%	77 K	3	1	EpicPlay	✗
26	Iobit Information Technology	✓	18	8	6%	73 K	3	1	Advanced SystemCare	✗
27	DT Soft	✗	14	2	22%	68 K	2	1	DAEMON Tools	✗
28	Innovative Apps	✗	14	1	68%	60 K	7	0	Supreme Savings	✓
29	Woolik Technologies	✗	13	9	70%	50 K	4	1	Woolik Search Tool	✓
30	Visual Software Systems	✓	22	12	62%	42 K	5	3	VisualBee	✗

Table 7: Top 30 advertiser clusters by installation base. For each publisher cluster it shows: whether we found an affiliate program (Aff), the in-degree (IN), out-degree (OD), detection ratio (DR), installation base (Hosts), number of parent PPI services (PPPI), number of child PPI services (CPPI), the main product advertised, and whether that product is a browser add-on (BAO) including toolbars, extensions, sidebars, and browser helper objects.

Event Type	Count
All PUP downloads	40.1M
Unsigned parent	11.5M
Signed parent	28.6M
Benign parent	7.4M
PUP parent	21.2M
PPI	7.3M
Adv. affiliate program	5.5M

Table 8: Analysis of PUP download events.

PC Speed Maximizer, Advanced System Care, DAEMON Tools). These rogueware try to convince users to buy the license for the full version. We also observe multimedia tools (Free Studio, GOM Player), backup tools (MyPC Backup), game promotion (EpicPlay), compressors (BitZipper), and presentation tools (Visual Bee).

7 PUP Distribution Methods.

This section measures the distribution of PUP through PPI services and affiliate programs. The relevant data is provided in Table 8. From the 90 M events with parent information, we first find the events with child files that are signed by PUP publishers (40.1M events). Then, we investigate the parents that installed them. In 28.6M (71%) of these events, parents were signed, therefore allowing us to go further in our search for finding the parents who are PPIs. 7.4M (35%) of these parents correspond to Web browsers and other benign download programs such as BitTorrent clients and Dropbox. The remaining 21.2M (65%) events have a PUP parent. This indicates that the majority of PUP is installed by other PUP. In particular, for 7.3M out of 21.2M events (34%)

with PUP parent, the parent corresponds to one of the 24 PPI services identified in Table 6. And, for another 5.5M (26%) events the parent corresponds to one of the 21 affiliate programs identified in Section 6. From these statistics, we can conclude that PUPs are generally installed by other PUPs and moreover, over 25% of the PUP download events are sourced by PPI services, and another 19% by advertisers with affiliate programs.

8 PUP–Malware Relationships

We are interested in understanding if there is any form of relationship between PUP and malware and if malware uses the PPI services we identified. In particular we would like to measure the percentage of PUP that installs malware or is installed by malware. Here, the obvious challenge is to accurately label malware in the WINE dataset. While the majority of properly signed executables flagged by AV engines are PUP, unsigned executables flagged by AV engines can be PUP or malware and there are a few malware that are signed.

To address these issues, we use AVClass, a recently released malware labeling tool [58]. Given the VT reports of a large number of executables, AVClass addresses the most important challenges in extracting malware family information from AV labels: label normalization, generic token detection, and alias detection. For each sample, it outputs a ranking of the most likely family names ranked by the number of AV engines assigning that family to the sample. Since AV labels can be noisy [4], we focus on executables for which the top family AVClass outputs is in a precomputed list of 70 malware families that includes prevalent families such as zbot, zeroaccess, reveton, virut, sality, shylock, and vobfus. Clearly, our methodology is not 100% accurate, but allows us to gain insight on the relationships between malware and PUP.

PUP downloading malware. One way malware authors could relate to PUP could be by signing up as advertisers to PPI services to distribute their malware. To identify such cases, we look for PUP publishers that download executables from one of the 70 malware families considered. What we have found out is that there is a link between 71 of the PUP publisher clusters to malware. Those publishers distribute malware from 40 families through 5,586 download events. Out of those 71 clusters, 11 are classified as PPI services in Section 6. Those PPI services generate 35% of the 5,586 malware downloads by PUP. For example, Perion Network, the most popular PPI service, downloads instances of zbot, shylock, and andromeda trojans. We also observe at the

end of 2013 iBaro downloading instances of sefnit click-fraud malware as reported by TrendMicro [38]. Clearly, 5,586 downloads is a low number, which may indicate that malware favors silent distribution vectors and that PPI services are careful to avoid malware to preserve their reputation towards security vendors. We only observe occasional events spread amongst multiple PPI services, possibly due to insufficient checks by those PPI services. Another factor of influence may be that installs through these PPIs can be an order of magnitude more expensive than those from silent PPIs, as shown in Section 2.1.

Malware downloading PUP. Malware authors could also sign up as affiliate publishers to PPI services to monetize the compromised machines by selling installs. To capture this behavior, we analyzed PUP downloaded by samples from the 70 malware families considered. We found 11K downloads by malware from 25 families. These malware samples downloaded executables from 98 PUP publisher clusters. 88% of these downloads were generated by 3 malware families: vobfus, badur, and delf. 7 of the 98 PUP publisher clusters belong to the PPI services category. For example, we observe zeroaccess installing files from the DomaIQ PPI service. Overall, malware downloading PUP is a more common event than PUP downloading malware, but still rare, affecting only 0.03% of all events where PUP is downloaded.

The conclusion of this analysis is that while PUP–malware interactions exist, they are not prevalent and malware distribution seems disjoint from PUP distribution. Observed malware–PPI service interactions do not focus on a few misbehaving PPI services, but rather seem to occasionally affect many PPI services.

9 Domain Analysis

In this section we analyze the 1.1 M events that contain a URL, and in particular the domains (ESLDs) in those URLs. The events that contain a URL allow us to identify publishers that download from and are downloaded from a domain. Note that the domains we extract from this dataset are used for hosting and distributing executables and do not cover all of the domains used by PUP. We identify 3 main types of domains from our analysis:

- **File lockers.** Cloud storage services used for backup or sharing executables between users. They exhibit a high number of client publishers being downloaded from them, most of which are benign (e.g., Microsoft, Adobe, AutoDesk). These ESLDs also host a front-end website for users.

- **Download portals.** They also distribute programs from a high number of publishers, predominantly free software publishers and their own PPI services. They also host a front-end website.
- **PPI services.** Used by PPI services to host their wrappers and advertised programs. These ESLDs do not host a front-end website as they are accessed by PPI installers, rather than humans.

Rank by downloaded publishers. Table 9 shows the top 20 ESLDs by number of child publishers signing files downloaded from that ESLD. The 4 tick-mark columns classify the domain as file locker (FL), download portal (DP), PPI service (PPI), or other (Oth). Of the 20 ESLDs, 15 correspond to file lockers, 2 to download portals, and another 2 to PPI services. The remaining domain is `file.org`, a portal where users can enter a file extension to find a tool that can open files with that extension. The publisher behind this portal uses it to promote its own free file viewer tool, which is offered as the best tool to handle over 200 file extensions.

If we give a vote to the top 3 publishers downloaded from each of the 15 file lockers (45 votes), Microsoft gets 13, Adobe 11, Cyberlink 4, and AutoDesk 3. The rest are popular benign publishers such as Ubisoft, VMWare, and Electronic Arts. Thus, file lockers predominantly distribute software from reputable publishers.

For the two download portals, the publishers downloaded from them correspond to their own PPI service (i.e., bundles signed by “CBS Interactive” from `cnet.com`), free software publishers, and PPI services. For `edgecastcdn.net` all 67 publishers are part of the same PPI service run by the Yontoo group. The domain `d3d6wi7c7pa6m0.cloudfront.net` belongs to the Adknowledge PPI service and distributes their advertiser programs. Among those advertiser programs we observe bundles signed by other PPI services, which may indicate arbitrageurs who try to take advantage of pricing differentials among PPI services [7].

Rank by downloads. Table 10 ranks the top 20 domains by number of downloads. It shows the ESLD, the type (file locker, download portal, PPI service, advertiser, other), the cluster that owns the domain, the number of downloads, the number of publishers of the downloaded executables, and the number of distinct files downloaded. We label each domain as belonging to the cluster that signs most executables downloaded from the domain. The publisher in the other category is Frostwire, which distributes a popular free BitTorrent client.

ESLD	FL	DP	PPI	Oth	Pub
uploaded.net	✓				366
cnet.com		✓			142
extabit.com	✓				128
share-online.biz	✓				125
4shared.com	✓				120
rapidgator.net	✓				90
depositfiles.com	✓				76
mediafire.com	✓				73
edgecastcdn.net			✓		67
chip.de		✓			53
zippyshare.com	✓				49
uloz.to	✓				48
file.org				✓	47
putlocker.com	✓				47
d3d6wi7c7pa6m0.cf			✓		44
turbobit.net	✓				44
freakshare.com	✓				41
rapidshare.com	✓				40
ddlstorage.com	✓				38
bitshare.com	✓				38

Table 9: Top 20 ESLDs by number of distinct publishers of downloaded executables. FL means file locker, DP download portal, PPI pay-per-install service, and Oth other. For brevity, `d3d6wi7c7pa6m0.cf` stands for `d3d6wi7c7pa6m0.cloudfront.net`.

Table 10 shows that PPI domains dominate in terms of downloads, but distribute a smaller number of child publishers compared to file lockers and download portals that dominate Table 9. It also shows that it is possible to link download domains to the publishers that own them based on the signature of files they distribute, despite the domains being typically registered by privacy protection services.

10 Discussion

Unsigned PUP. Our work focuses on signed PUP executables based on the prior observation that most signed samples flagged by AV engines are PUP [35]. However, this means that we will miss PUP publishers if they distribute only unsigned executables. Also, our PUP prevalence measurements are only a lower bound since there may be hosts with only unsigned PUP installed. In concurrent work, Thomas et al. [65] infiltrate 4 PPI services observing that only 58% of the advertiser software they distribute is signed. Thus, we could be missing as much as 42% of PUP software, but we expect a much smaller number of hosts will only have unsigned PUP installed.

ESLD	FL	DP	PPI	Ad	Oth	Cluster	Downl.	Pub.	Children
conduit.com			✓			Perion Network	138,480	2	727
edgecastcdn.net			✓			Yontoo	106,449	67	1,148
frostwire.com				✓		Frostwire	53,592	1	2,511
ask.com				✓		Ask	40,939	6	125
imgfarm.com				✓		Mindspark	26,498	6	3,209
ilivid.com				✓		Bandoo Media	25,429	5	905
conduit-services.com			✓			Perion Network	21,149	8	1,345
adpk.s3.amazonaws.com				✓		Adpeak	14,513	2	36
airdwnlds.com			✓			Air Software	14,342	1	13,389
ncapponline.info			✓			Web Pick	13,974	11	13,252
uploaded.net	✓					Cyando	10,886	366	7,816
storebox1.info			✓			Web Pick	10,109	13	9,561
oi-installer9.com			✓			Adknowledge	8,360	4	7,892
4shared.com	✓			✓		4shared	8,222	120	5,649
systweak.com				✓		Systweak	8,104	4	509
mypcbackup.com				✓		JDI Backup Limited	7,837	1	43
greatfilesarey.asia			✓			Web Pick	7,699	8	7,296
incredimail.com			✓			Perion Network	7,408	3	2,571
softonic.com		✓				Softonic	6,980	36	3,869
nicdls.com			✓			Tuguu	6,908	14	1,704

Table 10: Top ESLDs by number of downloads from them. The two rightmost columns are the number of publishers and files of the downloads.

Affiliate publisher analysis. We have classified publisher clusters as PPI services and advertisers, but we have not examined affiliate publisher clusters. One challenge with affiliate publishers is that when distribution happens through a stand-alone PPI installer (rather than bundles) both the advertiser program and the affiliate publisher program may appear as children of the PPI service in the publisher graph. It may be possible to measure the number of affiliates for some PPI services by analyzing URL parameters of download events. We leave this analysis to future work.

Other distribution models. We have examined PUP distribution through PPI services and advertiser affiliate programs. However, other distribution models exist. These include bilateral distribution agreements between two parties (e.g., Oracle’s Java distributing the Ask toolbar [34]) and pre-installed PUP (e.g., Superfish on Lenovo computers [21]). We observe Superfish distributed through PPI services prior to the Lenovo agreement, which started in September 2014 after our analysis period had ended. We leave the analysis of such distribution models to future work.

Observation period. Our observation period covers 19 months from January 2013 to July 2014. Unfortunately, WINE did not include newer data at the time of our study. Thus, we miss newer PUP publishers that joined the ecosystem after our observation period. However, the

vast majority of PUP publishers examined are still alive at the time of writing.

Internet population. We have measured the installation base of PUP (and benign) publishers on WINE hosts. We have also estimated that our measured WINE population may be two orders of magnitude lower than that of hosts connected to the Internet. But, we concede that this estimation is rough and could be affected by different factors such as selection bias.

11 Related Work

PUP. Potentially unwanted programs have received little attention from academia. In 2005–2007 Edelman studied the deceptive installation methods by spyware and other unwanted software [19]. In 2012, Pickard and Mladinov [52] studied a PUP rogue anti-malware software concluding that while not malicious, it only detected 0.3% of the malware and its main purpose was convincing the user to pay the license. Recently, some works have hinted at the increased prevalence and importance of PUP. Thomas et al. [64] study ad injectors, a type of PUP that modifies browser sessions to inject advertisements, finding that 5% of unique daily IP addresses accessing Google are impacted. In follow up work, Jagpal et al. [33] design WebEval, a system to identify mali-

cious extensions at the core of ad injection. Kotzias et al. [35] analyze abuse in Windows Authenticode by analyzing 356K samples from malware feeds. They find that PUP has been quickly increasing feeds since 2010, that the vast majority of properly signed samples are PUP, and that PUP publishers use high file and certificate polymorphism to evade security tools and CA defenses such as identity validation and revocation.

In concurrent work, Thomas et al. [65] analyze the advertiser software distributed to US hosts by 4 PPI services (OutBrowse, Amonetize, OpenCandy, InstallMonetizer). They also use SafeBrowsing data to measure that PPI services drive over 60 million download events every week, nearly three times that of malware. Both works are complementary in their study of PPI services and measuring users affected by PUP. They use a top-to-bottom approach of infiltrating a few PPI services plus SafeBrowsing data, while we perform a bottom-to-top approach starting from files installed on end hosts. We analyze 19 months from January 2013 to July 2014, while they analyze 12 months from August 2015 to July 2016. By examining download events on 3.9M WINE hosts in different countries, our approach enables us to measure PUP prevalence and achieves a broader coverage of the PPI ecosystem. We observe 23 PPI services including 3 of the 4 in their study. The missing PPI service is InstallMonetizer, which distributes mostly unsigned installers.

Also in concurrent work, Nelms et al. [42] analyzed web-based social engineering attacks that use deceiving advertisements to convince users to download unwanted software. They find that most programs distributed this way are bundles of free software with PUP.

Malware distribution. Prior work has studied malware distribution through different vectors, which differs from our focus on PUP distribution. Moschuk et al. [40] crawl 18M URLs finding that 5.9% were drive-by downloads and 13.4% lead to spyware. Provos et al. [53] study the prevalence of distribution through drive-by downloads. Grier et al. [22] analyze the commoditization of drive-by downloads and compare malware distribution through different vectors, concluding that drive-by downloads dominate. Caballero et al. [7] study malware distribution through PPI services. The PPI services we study differ in that installations are not silent and are mostly used by PUP and benign software. Kwon et al. [36] recently use WINE data to investigate malware distribution through downloaders. Their work differs in that they do not distinguish malware from PUP and in that they analyze file download graphs for individual machines. Instead, we

analyze download relationships between publishers on aggregate over 3.9M machines over a 19 month time period, focusing on PUP distribution through PPI services and affiliate programs.

12 Conclusion

We have performed the first systematic study of PUP prevalence and its distribution through PPI services. By using AV telemetry comprising of 8 billion events on 3.9 million hosts over 19 months, we have found that over half (54%) of the examined hosts have PUP installed. The top PUP publishers are highly popular; the top PUP publisher ranks 15 amongst all software publishers (benign or not). We have built the publisher graph that captures the who-installs-who relationships between PUP publishers. We have identified that 65% of the PUP is installed by other PUP and that 24 PPI services distribute over 25% of the PUP and advertiser affiliate programs an additional 19%. We have examined the PUP-malware relationships finding 11K events where popular malware families install PUP for monetization and 5,586 events where PUP distributes malware. PUP-malware interactions are not prevalent and seem to occasionally affect most top PPI services. We conclude that PUP distribution is largely disjoint from malware distribution.

13 Acknowledgments

We thank Richard Rivera for his help with the clustering. This research was partially supported by the Regional Government of Madrid through the N-GREENS Software-CM project S2013/ICE-2731 and by the Spanish Government through the Dedetis Grant TIN2015-7013-R. All opinions, findings and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

References

- [1] 7install. <https://web.archive.org/web/20160306081435/http://7install.com/>.
- [2] AdGazelle. <http://adgazelle.com/>.
- [3] AirSoftware. <https://airinstaller.com/>.

- [4] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification And Analysis Of Internet Malware. In *International Symposium on Recent Advances in Intrusion Detection*, Queensland, Australia, September 2007.
- [5] BetterInstaller. <http://betterinstaller.somotoinc.com/>.
- [6] L. Bilge and T. Dumitras. Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World. In *ACM Conference on Computer and Communications Security*, 2012.
- [7] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security*, 2011.
- [8] CashMyLinks. <http://www.cashmylinks.com/>.
- [9] Cinstaller. <http://cinstaller.com/>.
- [10] CodeFuel. 7 reasons codefuel beats all other pay per install companies, 2015. <http://www.codefuel.com/blog/7-reasons-perion-codefuel-beats-all-other-pay-per-install-companies/>.
- [11] ConversionAds. <https://web.archive.org/web/20160217095842/http://www.conversionads.com/>.
- [12] S. Davidoff. Interview with an adware author, 2009. <http://phlosecurity.org/2009/01/12/interview-with-an-adware-author>.
- [13] DomaiQ. <http://www.domaiq.com/en/>.
- [14] Download Admin. <https://web.archive.org/web/20140208040640/http://www.downloadadmin.com/>.
- [15] Download.com. <http://www.download.com/>.
- [16] T. Dumitraş and D. Shou. Toward a Standard Benchmark for Computer Security Research: The Worldwide Intelligence Network Environment (WINE). In *EuroSys Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, April 2011.
- [17] T. Dumitras and P. Efstathopoulos. The Provenance Of Wine. In *European Dependable Computing Conference*, May 2012.
- [18] EarnPerInstall. <https://web.archive.org/web/20160419013909/http://www.earnperinstall.com/>.
- [19] B. Edelman. Spyware Installation Methods. <http://www.benedelman.org/spyware/installations/>.
- [20] M. Geary. Adknowledge apps distribution opportunities, 2013. <http://ppitalk.com/showthread.php/49-Adknowledge-Apps-Distribution-Opportunities>.
- [21] D. Goodin. Lenovo pcs ship with man-in-the-middle adware that breaks https connections, 2015. <http://arstechnica.com/security/2015/02/lenovo-pcs-ship-with-man-in-the-middle-adware-that-breaks-https-connections/>.
- [22] Grier et al. Manufacturing Compromise: The Emergence Of Exploit-as-a-service. In *ACM Conference on Computer and Communications Security*, Raleigh, NC, October 2012.
- [23] GuppyGo. <http://www.guppygo.com/>.
- [24] Installaxy. <https://web.archive.org/web/20151105011933/http://installaxy.com/>.
- [25] InstallBay. <http://www.visibay.com/installbay>.
- [26] InstallCore. <https://www.installcore.com/>.
- [27] InstalleRex. <https://installerex.com/>.
- [28] Installertech. <http://www.installertech.com/>.
- [29] InstallMonetizer. <http://www.installmonetizer.com/>.
- [30] InstallMonster. <http://installmonster.ru/en>.
- [31] installPath. <http://www.installpath.com>.
- [32] Internet Archive WayBack Machine. <https://archive.org/web/>.
- [33] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and Lessons from Three Years Fighting Malicious Extensions. In *USENIX Security Symposium*, 2015.

- [34] O. Java. What are the Ask Toolbars? https://www.java.com/en/download/faq/ask_toolbar.xml.
- [35] P. Kotzias, S. Matic, R. Rivera, and J. Caballero. Certified PUP: Abuse in Authenticode Code Signing. In *ACM Conference on Computer and Communication Security*, 2015.
- [36] B. J. Kwon, J. Mondal, J. Jang, L. Bilge, and T. Dumitras. The Dropper Effect: Insights into Malware Distribution with Downloader Graph Analytics. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.
- [37] Mediakings. <https://web.archive.org/web/20140517213640/http://mediakings.com/>.
- [38] T. Micro. On the actors behind mevade/sefnit, 2014. <http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-on-the-actors-behind-mevade-sefnit.pdf>.
- [39] Microsoft. Windows authenticode portable executable signature format, Mar. 21 2008. http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx.
- [40] A. Moschuk, T. Bragin, S. D. Gribble, and H. Levy. A Crawler-based Study of Spyware in the Web. In *Network and Distributed System Security Symposium*, San Diego, CA, 2006.
- [41] NativeX. <http://nativex.com/>.
- [42] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad. Towards Measuring and Mitigating Social Engineering Malware Download Attacks. In *USENIX Security Symposium*, August 2016.
- [43] Net Cash Revenue. <http://netcashrevenue.com/>.
- [44] Nosibay. <http://www.nosibay.com/>.
- [45] Oneinstaller. <https://web.archive.org/web/20150220020855/http://oneinstaller.com/>.
- [46] Open Candy. <http://opencandy.com/>.
- [47] Opsiwat Antivirus and Threat Report, January 2014. <https://www.opswat.com/resources/reports/antivirus-january-2014.org/>.
- [48] PayPerInstall. <http://payperinstall.com/>.
- [49] Perinstallbox. <http://www.setupbundle.com/index.php>.
- [50] PerInstallBucks. <https://perinstallbucks.com/>.
- [51] PerInstallCash. <http://www.perinstallcash.com/>.
- [52] C. Pickard and S. Miladinov. Rogue software: Protection against potentially unwanted applications. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 1–8. IEEE, 2012.
- [53] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your Iframes Point To Us. In *USENIX Security Symposium*, San Jose, CA, July 2008.
- [54] Public Suffix List. <https://publicsuffix.org/>.
- [55] Purebits. <http://purebits.net/>.
- [56] RevenueHits. <https://web.archive.org/web/20130805140617/http://www.revenuehits.com/>.
- [57] RevenYou. <http://www.revenyou.com/>.
- [58] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. AVClass: A Tool for Massive Malware Labeling. In *International Symposium on Research in Attacks, Intrusions and Defenses*, September 2016.
- [59] P. Security. Malware still generated at a rate of 160,000 new samples a day in Q2 2014. <http://www.pandasecurity.com/mediacenter/press-releases/malware-still-generated-rate-160000-new-samples-day-q2-2014/>.
- [60] Smart WebAds. <http://www.smartwebads.com/>.
- [61] Softonic. www.softonic.com.
- [62] Solimba. <https://solimba.com/>.

- [63] Sterkly. <http://www.sterkly.com/>.
- [64] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *IEEE Symposium on Security and Privacy*, May 2015.
- [65] K. Thomass, J. A. E. Crespo, R. Rastil, J.-M. Picodi, L. Ballard, M. A. Rajab, N. Provos, E. Bursztein, and D. Mccoy. Investigating Commercial Pay-Per-Install and the Distribution of Unwanted Software. In *USENIX Security Symposium*, Aug. 2016.
- [66] Verti. <http://www.vertitechnologygroup.com>.
- [67] VirusTotal. <http://www.virustotal.com/>.
- [68] G. Wicherksi. pehash: A novel approach to fast malware clustering. In *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

A Additional Results

Rank	Publisher	Samples	
1	Popeler System	326,530	13.2%
2	Daniel Hareuveni	138,159	5.6%
3	Start Now	117,930	4.8%
4	Mail.Ru	117,920	4.8%
5	Softonic International	69,233	2.8%
6	Bon Don Jov	68,937	2.8%
7	Stepan Rybin	68,390	2.8%
8	WeDownload	66,332	2.7%
9	Payments Interactive	41,128	1.7%
10	Tiki Taka	37,072	1.5%
11	Stanislav Kabin	36,893	1.5%
12	Safe Software	36,602	1.5%
13	Vetaform Developments	36,001	1.5%
14	Outbrowse	35,832	1.4%
15	appbundler.com	34,895	1.4%
16	Rodion Veresev	34,696	1.4%
17	Mari Mara	31,031	1.3%
18	Firseria	29,940	1.2%
19	Give Away software	26,541	1.1%
20	Jelbrus	23,457	0.9%

Table 11: Top 20 publishers in the feed of 11M samples by number of samples and percentage over all samples signed and flagged by at least 4 AV engines.

#	PPI Service	Reseller
1	AdGazelle [2]	
2	EarnPerInstall [18]	
3	GuppyGo [23]	
4	Installaxy [24]	✓
5	InstallMonetizer [29]	
6	MediaKings [37]	
7	NetCashRevenue [43]	✓
8	PayPerInstall [48]	
9	PerInstallBox [49]	
10	PerInstallBucks [50]	✓
11	PerInstallCash [51]	
12	PureBits [55]	✓

Table 12: PPI services found through manual analysis on PPI forums and other Internet resources that are not present in our dataset. The reseller data comes from [65].

UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware

Amin Kharraz <i>Northeastern University</i> <i>mkharraz@ccs.neu.edu</i>	Sajjad Arshad <i>Northeastern University</i> <i>arshad@ccs.neu.edu</i>	Collin Mulliner <i>Northeastern University</i> <i>collin@mulliner.org</i>
William Robertson <i>Northeastern University</i> <i>wkr@ccs.neu.edu</i>	Engin Kirda <i>Northeastern University</i> <i>ek@ccs.neu.edu</i>	

Abstract

Although the concept of ransomware is not new (i.e., such attacks date back at least as far as the 1980s), this type of malware has recently experienced a resurgence in popularity. In fact, in the last few years, a number of high-profile ransomware attacks were reported, such as the large-scale attack against Sony that prompted the company to delay the release of the film “The Interview.” Ransomware typically operates by locking the desktop of the victim to render the system inaccessible to the user, or by encrypting, overwriting, or deleting the user’s files. However, while many generic malware detection systems have been proposed, none of these systems have attempted to specifically address the ransomware detection problem.

In this paper, we present a novel dynamic analysis system called UNVEIL that is specifically designed to detect ransomware. The key insight of the analysis is that in order to mount a successful attack, ransomware must tamper with a user’s files or desktop. UNVEIL automatically generates an artificial user environment, and detects when ransomware interacts with user data. In parallel, the approach tracks changes to the system’s desktop that indicate ransomware-like behavior. Our evaluation shows that UNVEIL significantly improves the state of the art, and is able to identify previously unknown evasive ransomware that was not detected by the anti-malware industry.

1 Introduction

Malware continues to remain one of the most important security threats on the Internet today. Recently, a specific form of malware called ransomware has become very popular with cybercriminals. Although the concept of ransomware is not new – such attacks were registered as far back as the end of the 1980s – the recent success of ransomware has resulted in an increasing number of new

families in the last few years [7, 20, 21, 44, 46]. For example, CryptoWall 3.0 made headlines around the world as a highly profitable ransomware family, causing an estimated \$325M in damages [45]. As another example, the Sony ransomware attack [27] received large media attention, and the U.S. government even took the official position that North Korea was behind the attack.

Ransomware operates in many different ways, from simply locking the desktop of the infected computer to encrypting all of its files. Compared to traditional malware, ransomware exhibits behavioral differences. For example, traditional malware typically aims to achieve stealth so it can collect banking credentials or keystrokes without raising suspicion. In contrast, ransomware behavior is in direct opposition to stealth, since the entire point of the attack is to openly notify the user that she is infected.

Today, an important enabler for behavior-based malware detection is dynamic analysis. These systems execute a captured malware sample in a controlled environment, and record its behavior (e.g., system calls, API calls, and network traffic). Unfortunately, malware detection systems that focus on stealthy malware behavior (e.g., suspicious operating system functionality for keylogging) might fail to detect ransomware because this class of malicious code engages in activity that appears similar to benign applications that use encryption or compression. Furthermore, these systems are currently not well-suited for detecting the specific behaviors that ransomware engages in, as evidenced by misclassifications of ransomware families by AV scanners [10, 39].

In this paper, we present a novel dynamic analysis system that is designed to analyze and detect ransomware attacks and model their behaviors. In our approach, the system automatically creates an artificial, realistic execution environment and monitors how ransomware interacts with that environment. Closely monitoring process interactions with the filesystem allows the system to precisely characterize cryptographic ransomware behavior.

In parallel, the system tracks changes to the computer’s desktop that indicates ransomware-like behavior. The key insight is that in order to be successful, ransomware will need to access and tamper with a victim’s files or desktop. Our automated approach, called UNVEIL, allows the system to analyze many malware samples at a large scale, and to reliably detect and flag those that exhibit ransomware-like behavior. In addition, the system is able to provide insights into how the ransomware operates, and how to automatically differentiate between different classes of ransomware.

We implemented a prototype of UNVEIL in Windows on top of the popular open source malware analysis framework Cuckoo Sandbox [13]. Our system is implemented through custom Windows kernel drivers that provide monitoring capabilities for the filesystem. Furthermore, we added components that run outside the sandbox to monitor the user interface of the target computer system.

We performed a long-term study analyzing 148,223 recent general malware samples in the wild. Our large-scale experiments show that UNVEIL was able to correctly detect 13,637 ransomware samples from multiple families in live, real-world data feeds with no false positives. Our evaluation also suggests that current malware analysis systems may not yet have accurate behavioral models to detect different classes of ransomware attacks. For example, the system was able to correctly detect 7,572 ransomware samples that were previously unknown and undetected by traditional AVs, but belonged to modern file locker ransomware families. UNVEIL was also able to detect a new type of ransomware that had not previously been reported by any security company. This ransomware also did not show any malicious activity in a modern sandboxing technology provided by a well-known anti-malware company, while showing heavy file encryption activity when analyzed by UNVEIL.

The high detection rate of our approach suggests that UNVEIL can complement current malware analysis systems to quickly identify new ransomware samples in the wild. UNVEIL can be easily deployed on any malware analysis system by simply attaching to the filesystem driver in the analysis environment.

In summary, this paper makes the following contributions:

- We present a novel technique to detect ransomware known as *file lockers* that targets files stored on a victim’s computer. Our technique is based on monitoring system-wide filesystem accesses in combination with the deployment of automatically-generated artificial user environments for triggering ransomware.
- We present a novel technique to detect ransomware

known as *screen lockers*. Such ransomware prevents access to the computer system itself. Our technique is based on detecting locked desktops using dissimilarity scores of screenshots taken from the analysis system’s desktop before, during, and after executing the malware sample.

- We performed a large-scale evaluation to show that our approach can effectively detect ransomware. We automatically detected and verified 13,637 ransomware samples from a dataset of 148,223 recent general malware. In addition, we found one previously unknown ransomware sample that does not belong to any previously reported family. Our evaluation demonstrates that our technique works well in practice (achieving a true positive [TP] rate 96.3% at zero false positives [FPs]), and is useful in automatically identifying ransomware samples submitted to analysis and detection systems.

The rest of the paper is structured as follows. In Section 2, we briefly present background information and explain different classes of ransomware attacks. In Section 3, we describe the architecture of UNVEIL and explain our detection approaches for multiple types of ransomware attacks. In Section 4, we provide more details about our dynamic analysis environment. In Section 5, we present the evaluation results. Limitations of the approach are discussed in Section 6, while Section 7 presents related work. Finally, Section 8 concludes the paper.

2 Background

Ransomware, like other classes of malware, uses a number of strategies to evade detection, propagate, and attack users. For example, it can perform multi-infection or process injection, exfiltrate the user’s information to a third party, encrypt files, and establish secure communication with C&C servers. Our detection approach assumes that ransomware samples can and will use all of the techniques that other malware samples may use. In addition, our system assumes that successful ransomware attacks perform one or more of the following activities.

Persistent desktop message. After successfully performing a ransomware infection, the malicious program typically displays a message to the victim. This “ransom note” informs the users that their computer has been “locked” and provides instructions on how to make a ransom payment to restore access. This ransom message can be generated in different ways. A popular technique is to call dedicated API functions (e.g., `CreateDesktop()`) to create a new desktop and make it the default config-

uration to lock the victim out of the compromised system. Malware writers can also use HTML or create other forms of persistent windows to display this message. Displaying a persistent desktop message is a classic action in many ransomware attacks.

Indiscriminate encryption and deletion of the user’s private files. A crypto-style ransomware attack lists the victim’s files and aggressively encrypts any private files it discovers. Access is restricted by withholding the decryption key. Encryption keys can be generated locally by the malware on the victim’s computer, or remotely on C&C servers, and then delivered to the compromised computer. An attacker can use customized destructive functions, or Windows API functions to delete the original user’s files. The attacker can also overwrite files with the encrypted version, or use secure deletion via the Windows Secure Deletion API.

Selective encryption and deletion of the user’s private files based on certain attributes (e.g., size, date accessed, extension). In order to avoid detection, a significant number of ransomware samples encrypt a user’s private files selectively. In the simplest form, the ransomware sample can list the files based on the access date. In more sophisticated scenarios, the malware could also open an application (e.g., `word.exe`) and list recently accessed files. The sample can also inject malicious code into any Windows application to obtain this type of information (e.g., directly reading process memory).

In this work, we address all of these scenarios where an adversary has already compromised a system, and is able to launch arbitrary ransomware-related operations on the user’s files or desktop.

3 UNVEIL Design

In this section, we describe our techniques for detecting multiple classes of ransomware attacks. We refer the reader to Section 4 for details on the implementation details of the prototype.

3.1 Detecting File Lockers

We first describe why our system creates a unique, artificial user environment in each malware run. We then present the design of the filesystem activity monitor and describe how UNVEIL uses the output of the filesystem monitor to detect ransomware.

3.1.1 Generating Artificial User Environments

Protecting malware analysis environments against fingerprinting techniques is non-trivial in a real-world deployment. Sophisticated malware authors exploit static fea-

tures inside analysis systems (e.g., name of a computer) and launch reconnaissance-based attacks [31] to fingerprint both public and private malware analysis systems.

The static features of analysis environments can be viewed as the Achilles’ heel of malware analysis systems. One static feature that can have a significant impact on the effectiveness of the malware analysis systems is the user data that can be effectively used to fingerprint the analysis environment. That is, even on bare-metal environments where classic tricks such as virtualization checks are not possible, an unrealistic looking user environment can be a telltale sign that the code is running in a malware analysis system.

Intuitively, a possible approach to address such reconnaissance attacks is to build the user environment in such a way that the user data is valid, real, and non-deterministic in each malware run. These automatically-generated user environments serve as an “enticing target” to encourage ransomware to attack the user’s data while at the same time preventing the possibility of being recognized by adversaries.

In practice, generating a user environment is a non-trivial problem, especially if this is to be done automatically. This is because the content generator should not allow the malware author to fingerprint the automatically-generated user content located in the analysis environment, and also determine that it does not belong to a real user. We elaborate on how we automatically generate an artificial – yet realistic – user environment for ransomware in each malware run in Section 4.1.

3.1.2 Filesystem Activity Monitor

The filesystem monitor in UNVEIL has direct access to data buffers involved in I/O requests, giving the system full visibility into all filesystem modifications. Each I/O operation contains the process name, timestamp, operation type, filesystem path and the pointers to the data buffers with the corresponding entropy information in read/write requests. The generation of I/O requests happens at the lowest possible layer to the filesystem. For example, there are multiple ways to read, write, or list files in user-/kernel-mode, but all of these functions are ultimately converted to a sequence of I/O requests. Whenever a user thread invokes an I/O API, an I/O request is generated and is passed to the filesystem driver. Figure 1 shows a high-level design of UNVEIL in the Windows environment.

UNVEIL’s monitor sets callbacks on all I/O requests to the filesystem generated on behalf of any user-mode processes. We note that for UNVEIL operations, it is desirable to only set one callback per I/O request for performance reasons, and that this also maintains full visibility into I/O operations. In UNVEIL, user-mode process in-

teractions with the filesystem are formalized as access patterns. We consider access patterns in terms of I/O traces, where a trace T is a sequence of t_i such that

$$t_i = \langle P, F, O, E \rangle,$$

P is the set of user-mode processes,

F is the set of available files,

O is the set of I/O operations, and

E is the entropy of read or write data buffers.

For all of the file locker ransomware samples that we studied, we empirically observed that these samples issue I/O traces that exhibit distinctive, repetitive patterns. This is due to the fact that these samples each use a single, specific strategy to deny access to the user’s files. This attack strategy is accurately reflected in the form of I/O access patterns that are repeated for each file when performing the attack. Consequently, these I/O access patterns can be extracted as a distinctive I/O fingerprint for a particular family. We note that our approach mainly considers write or delete requests. We elaborate on extracting I/O access patterns per file in Section 3.1.2.

I/O Data Buffer Entropy. For every read and write request to a file captured in an I/O trace, UNVEIL computes the entropy of the corresponding data buffer. Comparing the entropy of read and write requests to and from the same file offset serves as an excellent indicator of crypto-ransomware behavior. This is due to the common strategy to read in the original file data, encrypt it, and overwrite the original data with the encrypted version. The system uses Shannon entropy [30] for this computation. In particular, assuming a uniform random distribution of bytes in a data block d , we have

$$H(d) = - \sum_{i=1}^n \frac{\log_2 n}{n}.$$

Constructing Access Patterns. For each execution, after UNVEIL generates I/O access traces for the sample, it sorts the I/O access requests based on file names and request timestamps. This allows the system to extract the I/O access sequence for each file in a given run, and check which processes accessed each file. The key idea is that after sorting the I/O access requests per file, repetition can be observed in the way I/O requests are generated on behalf of the malicious process.

The particular detection criterion used by the system to detect ransomware samples is to identify write and delete operations in I/O sequences in each malware run. In a successful ransomware attack, the malicious process typically aims to encrypt, overwrite, or delete user files at some point during the attack. In UNVEIL, these I/O

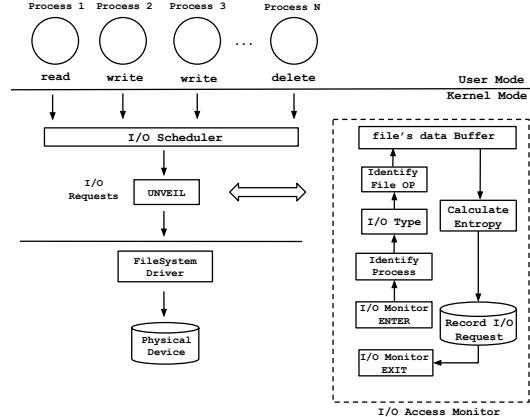


Figure 1: Overview of the design of I/O access monitor in UNVEIL. The module monitors system-wide filesystem accesses of user-mode processes. This allows UNVEIL to have full visibility into interactions with user files.

request patterns raise an alarm, and are detected as suspicious filesystem activity. We studied different file locker ransomware samples across different ransomware families. Our analysis shows that although these attacks can be very different in their attack strategies (e.g., evasion techniques, key generation, key management, connecting to C&C servers), they can be categorized into three main classes of attacks based on their access requests.

Figure 2 shows the high-level access patterns for multiple ransomware families we studied during our experiments. For example, the access pattern shown to the left is indicative of Cryptolocker variants that have varying key lengths and desktop locking techniques. However, its access pattern remains constant with respect to family variants. We observed the same I/O activity for samples in the CryptoWall family as well. While these families are identified as two different ransomware families, since they use the same encryption functions to encrypt files (i.e., the Microsoft CryptoAPI), they have similar I/O patterns when they attack user files.

As another example, in FileCoder family, the ransomware first creates a new file, reads data from a victim’s file, generates an encrypted version of the original data, writes the encrypted data buffer to the newly generated file, and simply unlinks the original user’s file (See Figure 2.2). In this class of file locker ransomware, the malware does not wipe the original file’s data from the disk. For attack approaches like this, victims have a high chance of recovering their data without paying the ransom. In the third approach (Figure 2.3), however, the ransomware creates a new encrypted file based on the original file’s data and then securely deletes the original file’s data using either standard Windows APIs or custom overwriting implementations (e.g., such as Cryp-

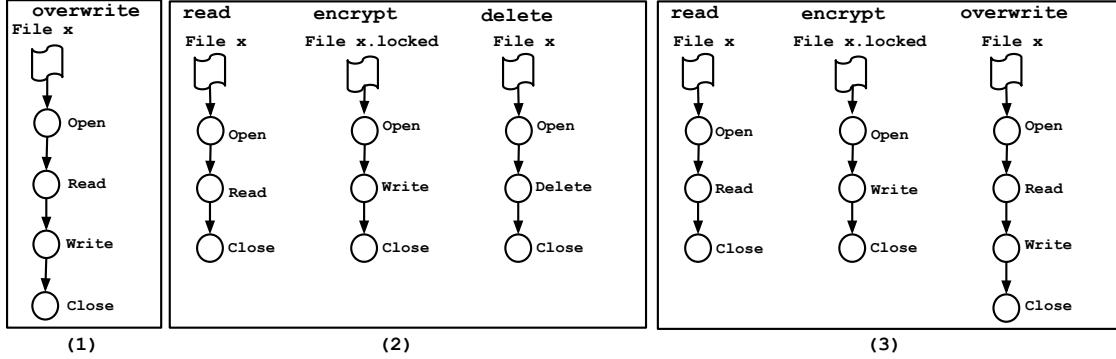


Figure 2: Strategies differ across ransomware families with respect to I/O access patterns. (1) Attacker overwrites the users’ file with an encrypted version; (2) Attacker reads, encrypts and deletes files without wiping them from storage; (3) Attacker reads, creates a new encrypted version, and securely deletes the original files by overwriting the content.

Vault family).

3.2 Detecting Screen Lockers

The second core component of UNVEIL is aimed at detecting screen locker ransomware. The key insight behind this component is that the attacker must display a ransom note to the victim in order to receive a payment. In most cases, the message is prominently displayed, covering a significant part, or all, of the display. As this ransom note is a virtual invariant of ransomware attacks, UNVEIL aims to automatically detect the display of such notes.

The approach adopted by UNVEIL to detect screen locking ransomware is to monitor the desktop of the victim machine, and to attempt to detect the display of a ransom note. Similar to Grier et al. [15], we take automatic screenshots of the analysis desktop before and after the sample is executed. The screenshots are captured from outside of the dynamic analysis environment to prevent potential tampering by the malware. This series of screenshots is analyzed and compared using image analysis methods to determine if a large part of the screen has suddenly changed between captures. However, smaller changes in the image such as the location of the mouse pointer, current date and time, new desktop icons, windows, and visual changes in the task bar should be rejected as inconsequential.

In UNVEIL, we measure the *structural similarity* (SSIM) [49] of two screenshots – before and after sample execution – by comparing local patterns of pixel intensities in terms of both luminance and contrast as well as the structure of the two images. Extracting structural information is based on the observation that pixels have strong inter-dependencies – especially when they are spatially close. These dependencies carry information about the structure of the objects in the image. After a successful ransomware attack, the display of the ransom note often

results in automatically identifiable changes in the structural information of the screenshot (e.g., a large rectangular object covers a large part of the desktop). Therefore, the similarity of the pre- and post-attack images decreases significantly, and can be used as an indication of ransomware.

In order to avoid false positives, UNVEIL only takes screenshots resulting from persistent changes (i.e., changes that cannot be easily dismissed through user interaction). The system first removes such transient changes (e.g., by automatically closing open windows) before taking screenshots of the desktop. Using this pre-processing step, ransomware-like applications that are developed for other purposes such as fake AV are safely categorized as non-ransomware samples.

UNVEIL also extracts the text within the area where changes in the structure of the image has occurred. The system extracts the text inside the selected area and searches for specific keywords that are highly correlated with ransom notes (e.g., <lock, encrypt, desktop, decryption, key>).

Given two screenshots X and Y , we define the structural similarity index of the image contents of local windows x_j and y_j as

$$\text{LocalSim}(x_j, y_j) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

where μ_x and μ_y are the mean intensity of x_j and y_j , and σ_x and σ_y are the standard deviation as an estimate of x_j and y_j contrast and σ_{xy} is the covariance of x_j and y_j . The local window size to compare the content of two images was set 8×8 . c_1 and c_2 are division stabilizer in the SSIM index formula [49]. We define the overall similarity between the two screenshots X and Y as the arithmetic mean of the similarity of the image contents x_j and y_j at the j^{th} local window where M is the number

of local windows of X and Y :

$$\text{ImgSim}(X, Y) = \frac{1}{M} \sum_{j=1}^M \text{LocalSim}(x_j, y_j).$$

Since the overall similarity is always on $[0, 1]$, the distance between X and Y is simply defined as

$$\text{Dist}(X, Y) = 1 - \text{ImgSim}(X, Y).$$

Finally, we define a similarity threshold τ_{sim} such that UNVEIL considers the sample a potential screen locking ransomware if

$$\text{Dist}(X, Y) > \tau_{\text{sim}}.$$

UNVEIL then extracts the text within the image and searches for ransomware-related words within the modified area. Applying the image similarity test with the best similarity threshold (see Section 5.2.2) gives us the highest recall with 100% precision for the entire dataset.

4 UNVEIL Implementation

In this section, we describe the implementation details of a prototype of UNVEIL for the Windows platform. We chose Windows for a proof-of-concept implementation because it is currently the main target of ransomware attacks. We elaborate on how UNVEIL automatically generates artificial, but realistic user environments for each analysis run, how the system-wide monitoring was implemented, and how we deployed the prototype of our system.

4.1 Generating User Environments

In each run, the user environment is made up of several forms of content such as digital images, videos, audio files, and documents that can be accessed during a user *Windows Session*. The user content is automatically-generated according to the following process:

For each file extension from a space of possible extensions, a set of files are generated where the number of files for each extension is sampled from a uniform random distribution for each sample execution. Each set of files collectively forms a *document space* for the sample execution environment. From a statistical perspective, document spaces generated for each sample execution should be indistinguishable from real user data. As an approximation to this ideal, randomly-selected numbers of files are generated per extension for each run according to the process described above.

In the following, we describe the additional properties that a document space should have in order to complicate programmatic approaches that ransomware samples can

potentially use to identify the automatically-generated user environment.

Valid Content. The user content generator creates real files with valid headers and content using standard libraries (e.g., `python-docx`, `python-pptx`, `OpenSSL`). Based on empirical observation, we created four file categories that a typical ransomware sample tries to find and encrypt: documents, keys and licenses, file archives, and media. Document extensions include `txt`, `doc(x)`, `ppt(x)`, `tex`, `xls(x)`, `c`, `pdf` and `py`. Keys and license extensions include `key`, `pem`, `crt`, and `cer`. Archive extensions include `zip` and `rar` files. Finally, media extensions include `jp(e)g`, `mp3`, and `avi`. For each sample execution, a subset of extensions are randomly selected and are used to generate user content across the system.

In order to generate content that appears meaningful, we collected approximately 100,000 sentences by querying 500 English words in Google. For each query, we collected the text from the first 30 search results to create a sentence list. We use the collected sentences to generate the content for the user files. We used the same technique to create a word list to give a name to the user files. The word list allows us to create files with variable name lengths that do not appear random. Clearly, the problem with random content and name generation (e.g., `xteyshfqb.docx`) is that the attacker could programmatically calculate the entropy of the file names and contents to detect content that has been generated automatically. Hence, by generating content that appears meaningful, we make it difficult for the attacker to fingerprint the system and detect our generated files.

File Paths. Note that the system is also careful to randomly generate the supposed victim’s directory structure. For example, directory names are also generated based on meaningful words. Furthermore, the system also associates files of certain types with standard locations in the Windows directory structure for those file types (e.g., the system does not create document files in a directory with image files, but rather under `My Documents`). The path length of user files is also non-deterministic and is generated randomly. In addition, each folder may have a set of sub-folders. Consequently, the generated paths to user files have variable depths relative to the root folder.

Time Attributes. Another non-determinism strategy used by our approach is to generate files with different *creation*, *modification*, and *access* times. The file time attributes are sampled from a distribution of likely timestamps when creating the file. When the system creates files with different time attributes, the time attributes of the containing folders are also updated automatically. In this case, the creation time of the folder is the minimum of all creation times of files and folders inside the folder, while the modification and access times are the maxi-

mum of the corresponding timestamps.

While we have not observed ransomware samples that have attempted to use fingerprinting heuristics of the content of the analysis environment, the nondeterminism strategies used by UNVEIL serve as a basis for making the analysis resilient to fingerprinting by design.

4.2 Filesystem Activity Monitor

Several techniques have been used to monitor sample filesystem activity in malware analysis environments. For example, filesystem activity can be monitored by hooking a list of relevant filesystem API functions or relevant system calls using the System Service Descriptor Table (SSDT). Unfortunately, these approaches are not suitable for UNVEIL’s detection approach for several reasons. First, API hooking can be bypassed by simply copying a DLL containing the desired code and dynamically loading it into the process’ address space under a different name. Stolen code [17, 19] and sliding calls [19] are other examples of API hooking evasion that are common in the wild. Furthermore, ransomware can use customized cryptosystems instead of the standard APIs to bypass API hooking while encrypting user files. Hooking system calls via the SSDT also has other technical limitations. For example, it is prevented on 64-bit systems due to Kernel Patch Protection (KPP). Furthermore, most SSDT functions are undocumented and subject to change across different versions of Windows.

Therefore, instead of API or system call hooking, UNVEIL monitors filesystem I/O activity using the Windows Filesystem Minifilter Driver framework [34], which is a standard kernel-based approach to achieving system-wide filesystem monitoring in multiple versions of Windows. The prototype consists of two main components for I/O monitoring and retrieving logs of the entire system with approximately 2,800 SLOC in C++. In Windows, I/O requests are represented by I/O Request Packets (IRPs). UNVEIL’s monitor sets callbacks on all I/O requests to the filesystem generated on behalf of user-mode processes. Basing UNVEIL’s filesystem monitor on a minifilter driver allows it to be located at the closest possible layer to the filesystem with access to nearly all objects of the operating system.

4.3 Desktop Lock Monitor

To identify desktop locking ransomware, screenshots are captured from outside of the dynamic analysis environment to prevent potential tampering by the malware. For dissimilarity testing, a python script implements the Structural Similarity Image Metric (SSIM) as described in Section 3.2. UNVEIL first converts the images to floating point data, and then calculates parameters such as

mean intensity μ using Gaussian filtering of the images’ contents. We also used default values ($k_1 = 0.01$ and $k_2 = 0.03$) to obtain the values of c_1 and c_2 to calculate the structural similarity score in local windows presented in Section 3.2.

The system also employs Tesseract-OCR [38], an open source OCR engine, to extract text from the selected areas of the screenshots. To perform the analysis on the extracted text within images, we collected more than 10,000 unique ransom notes from different ransomware families. We first clustered ransom notes based on the family type and the visual appearance of the ransom notes. For each cluster, we then extracted the ransom texts in the corresponding ransom notes and performed pre-filtering to remove unnecessary words within the text (e.g., articles, pronouns) to avoid obvious false positive cases. The result is a word list for each family cluster that can be used to identify ransom notes and furthermore label notes belonging to a known ransomware family.

5 Evaluation

We evaluated UNVEIL with two experiments. The goal of the first experiment is to demonstrate that the system can detect known ransomware samples, while the goal of the second experiment is to demonstrate that UNVEIL can detect previously unknown ransomware samples.

5.1 Experimental Setup

The UNVEIL prototype is built on top of Cuckoo Sandbox [13]. Cuckoo provides basic services such as sample submission, managing multiple VMs, and performing simple human interaction tasks such as simulating user input during an analysis. However, in principle, UNVEIL could be implemented using any dynamic analysis system (e.g., BitBlaze [5], VxStream Sandbox [37]).

We evaluated UNVEIL using 56 VMs running Windows XP SP3 on a Ganeti cluster based on Ubuntu 14.04 LTS. While Windows XP is not required by UNVEIL, it was chosen because it is well-supported by Cuckoo sandbox. Each VM had multiple NTFS drives. We took anti-evasion measures against popular tricks such as changing the IP address range and the MAC addresses of the VMs to prevent the VMs from being fingerprinted by malware authors. Furthermore, we permitted controlled access to the Internet via a filtered host-only adapter. In particular, the filtering allowed limited IRC, DNS, and HTTP traffic so samples could communicate with C&C servers. SMTP traffic was redirected to a local honeypot to prevent spam, and network bandwidth was limited to mitigate potential DoS attacks.

Family	Type	Samples
Cryptolocker	crypto	33 (1.5%)
CryptoWall	crypto	42 (2.0%)
CTB-Locker	crypto	77 (3.6%)
CrypVault	crypto	21 (1.0%)
CoinVault	crypto	17 (0.8%)
Filecoder	crypto	19 (0.9%)
TeslaCrypt	crypto	39 (1.8%)
Tox	crypto	71 (3.3%)
VirLock	locker	67 (3.2%)
Reveton	locker	501 (23.6%)
Tobfy	locker	357 (16.8%)
Urausy	locker	877 (41.3%)
Total Samples	-	2,121

Table 1: The list of ransomware families used in the first experiment.

The operating system image inside the malware analysis system included typical user data such as saved social networking credentials and a valid browsing history. For each operating system image, multiple users were defined to run the experiments. We also ran a script that emulated basic user activity while the malware sample was running on the system, such as launching a browser and navigating to multiple websites, or clicking on the desktop. This interaction was randomly-generated, but was constant across runs. Each sample was executed in the analysis environment for 20 minutes. As described in Sections 3.1 and 3.2, user environments were generated for each run, filesystem I/O traces were recorded, and pre- and post-execution screenshots were captured. After each execution, the VM was rolled back to a clean state to prevent any interference across executions. All experiments were performed according to well-established experimental guidelines [40] for malware experiments.

5.2 Ground Truth (Labeled) Dataset

In this experiment, we evaluated the effectiveness of UNVEIL on a labeled dataset, and ran different screen locker samples to determine the best threshold value τ_{sim} for the large-scale experiment.

We collected ransomware samples from public repositories [1, 3] and online forums that share malware samples [2, 32]. We also received labeled ransomware samples from two well-known anti-malware companies. In total, we collected 3,156 recent samples. In order to make sure that those samples were indeed active ransomware, we ran them in our test environment. We confirmed 2,121 samples to be active ransomware instances. After each run, we checked the filesystem activity of each sample for any signs of attacks on user data. If we did not see any malicious filesystem activity, we checked whether running the sample displayed a ransom note.

Table 1 describes the ransomware families we used in this experiment. We note that the dataset covers the majority of the current ransomware families in the wild. In

Run	OP	Proc	FName	Offset	Entropy
CryptoWall 3	read write ...	explorer.exe explorer.exe	document.cad document.cad	[0,4096) [0,4096)	5.21 7.04
CryptoWall 4	read write ... rename	explorer.exe explorer.exe explorer.exe	project.cad project.cad t67djkje.elkd8	[0,4096) [0,4096)	5.21 7.11

Table 2: An example of I/O access in UNVEIL for CryptoWall 3.0 and CryptoWall 4.0.

Application	OP	Description
CrypVault	read write ...	read low entropy buffer from original file write high entropy buffer to a new file
	write delete	overwrite the buffer of the original file read attributes, delete the original file
CryptoWall4	read write ... rename	read low entropy buffer overwrite with high entropy buffer read attributes, rename the files
SDelete	write ... delete	overwrite data buffer read attributes, delete the file
7-zip	read write ...	read data buffer from original file write data buffer to a new file

Table 3: I/O accesses for deletion and compression mechanisms in benign/malicious applications. Benign programs can generate I/O access requests similar to ransomware attacks, but since they are not designed to deny access to the original files, their I/O sequence patterns are different from ransomware attacks.

addition to the labeled ransomware dataset, we also created a dataset that consisted of non-ransomware samples. These samples were submitted to the Anubis analysis platform [16], and consisted of a collection of benign as well as malicious samples. We selected 149 benign executables including applications that have ransomware-like behavior such as secure deletion, encryption, and compression. A short list of these applications are provided in Table 5. We also tested 384 non-ransomware malware samples from 36 malware families to evaluate the false positive rate of UNVEIL. Table 2 shows an example of I/O traces for CryptoWall 3.0 and CryptoWall 4.0 where the victim’s file is first read and then overwritten with an encrypted version. The I/O access patterns of CryptoWall 4.0 samples to overwrite the content of the files are identical since they use the same cryptosystem. The main difference is that the filenames and extensions are modified with random characters, probably to minimize the chance of recovering the files based on their names in the Master File Table (MFT) in the NTFS filesystem.

5.2.1 Filesystem Activity of Benign Applications with Potential Ransomware-like Behavior

One question that arises is whether benign applications such as encryption or compression programs might generate similar I/O request sequences, resulting in false positives. Note that with benign applications, the original file content is treated carefully since the ultimate goal is to generate an encrypted version of the original file, and not to restrict access to the file. Therefore, the default mechanism in these applications is that the original files remain intact even after encryption or compression. If automatic deletion is deliberately activated by the user after the encryption, it can potentially result in a false positive (see Figure 2.2). However, in our approach, we assume that the usual default behavior is exhibited and the original data is preserved. We believe that this is a reasonable assumption, considering that we are building an analysis system that will mainly analyze potentially suspicious samples captured and submitted for analysis. Nevertheless, we investigated the I/O access patterns of benign programs, shown in Table 3. The I/O traces indicate that these programs exhibit distinguishable I/O access patterns as a result of their default behavior.

Benign applications might not necessarily perform encryption or deletion on user files, but can change the content of the files. For example, updating the content of a Microsoft PowerPoint file (e.g., embedding images and media) generates I/O requests similar to ransomware (see Figure 2.1). However, the key difference here is that such applications usually generate I/O requests for a single file at a time and repetition of I/O requests does not occur over multiple user files. Also, note that benign applications typically do not arbitrarily encrypt, compress or modify user files, but rather need sophisticated input from users (e.g., file names, keys, options, etc.). Hence, most applications would simply exit, or expect some input when executed in UNVEIL.

5.2.2 Similarity Threshold

We performed a precision-recall analysis to find the best similarity threshold τ_{sim} for desktop locking detection. The best threshold value to discriminate between similar and dissimilar screenshots should be defined in such a way that UNVEIL is able to detect screen locker ransomware while maintaining an optimal precision-recall rate. Figure 3 shows empirical precision-recall results when varying τ_{sim} . As the figure shows, with $\tau_{\text{sim}} = 0.32$, more than 97% of the ransomware samples across both screen and file locker samples are detected with 100% precision. In the second experiment, we used this similarity threshold to detect screen locker ransomware in a malware feed unknown to UNVEIL.

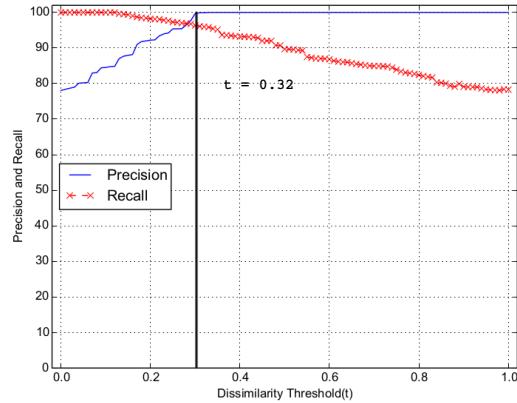


Figure 3: Precision-recall analysis of the tool. The threshold value $\tau_{\text{sim}} = 0.32$ gives the highest recall with 100% precision.

Evaluation	Results
Total Samples	148,223
Detected Ransomware	13,637 (9.2%)
Detection Rate	96.3%
False Positives	0.0%
New Detection	9,872 (72.2%)

Table 4: UNVEIL detection results. 72.2% of the ransomware samples detected by UNVEIL were not detected by any of AV scanners in VirusTotal at the time of the first submission. 7,572 (76.7%) of the newly detected samples were destructive file locker ransomware samples.

5.3 Detecting Zero-Day Ransomware

The main goal of the second experiment is to evaluate the accuracy of UNVEIL when applied to a large dataset of recent real-world malware samples. We then compared our detection results with those reported by AV scanners in VirusTotal.

This dataset was acquired from the daily malware feed provided by Anubis [16] to security researchers. The samples were collected from May 18th 2015 until February 12th 2016. The feed is generated from the Anubis submission queue, which is fed in turn by Internet users and security companies. Hence, before performing the experiment, we filtered the incoming Anubis samples by removing those that were not obviously executable (e.g., PDFs, images). After this filtering step, the dataset contained 148,223 distinct samples. Each sample was then submitted to UNVEIL to obtain I/O access traces and pre-/post-execution desktop image dissimilarity scores.

5.3.1 Detection Results

Table 4 shows the evaluation results of the second experiment. With the similarity threshold $\tau_{\text{sim}} = 0.32$, UNVEIL labeled 13,637 (9.2% of the dataset) samples in the Anu-

bis malware feed as being ransomware; these included both file locker and desktop locker samples.

Evaluation of False Positives. As we did not have a labeled ground truth in the second experiment, we cannot provide an accurate precision-recall analysis, and verifying the detection results is clearly challenging. For example, re-running samples while checking for false positives is not feasible in all cases since samples may have become inactive at the time of re-analysis (e.g., the C&C server might have been taken down).

Hence, we used manual verification of the detection results. That is, for the samples that were detected as screen locker ransomware, we manually checked the post-attack screenshots that were reported taken by UNVEIL. The combination of structural similarity test and the OCR technique to extract the text provides a reliable automatic detection for this class of ransomware. We confirmed that UNVEIL correctly reported 4,936 samples that delivered a ransom note during the analysis.

Recall that UNVEIL reports a sample as a file locker ransomware if the I/O access pattern follows one of the three classes of ransomware attacks described in Figure 2. For file locker ransomware samples, we used the I/O reports for each sample. We listed all the I/O activities on the first five user files during that run and looked for suspicious I/O activity such as requesting *write* and/or *delete* operations. Note that the detection approach used in UNVEIL is only based on the I/O access pattern. We do not check for changes in entropy in the detection phase and it is only used for our evaluation.

If we find multiple write or delete I/O requests to the first five generated user files and also a significant increase in the entropy between read and write data buffers at a given file offset, or the creation of new high entropy files, we confirmed the detection as a true positive. The creation of multiple new high entropy files based on user files is a reliable sign of ransomware in our tests. For example, the malware sample that uses secure deletion techniques may overwrite files with low entropy data. However, the malicious program first needs to generate an encrypted version of the original files. In any case, generating high entropy data raises an alarm in our evaluation.

By employing these two approaches and analyzing the results, we did not find any false positives. There were a few cases that had significant change in the structure of the images. Our closer investigation revealed that the installed program generated a large installation page, showed some unreadable characters in the window, and did not close even if the button was clicked (i.e., non-functional buttons). In another case, the program generated a large setup window, but it did not proceed due to a crash. These cases produce a higher dissimilarity

score than the threshold value. However, since the extracted text within those particular windows did not contain any ransomware-related contents, UNVEIL safely categorized them as being non-ransomware samples.

Evaluation of False Negatives. Determining false negative rates is a challenge since manually checking 148,223 samples is not feasible. In the following, we provide an *approximation* of false negatives for UNVEIL.

In our tests on the labeled dataset, false negatives mainly occurred in samples that make persistent changes on the desktop, but since the dissimilarity score of pre-/post-attack is less than $\tau_{\text{sim}} = 0.32$, it is not detected as ransomware by UNVEIL. Our analysis of labeled samples from multiple ransomware families (see Section 5.2) shows that these cases were mainly observed in samples with a similarity score between the interval [0.18, 0.32]. This is because for lower similarity scores, changes in the screenshots are negligible or small (e.g., Windows warning/error messages). Consequently, in order to increase the chance of catching false negative cases, we selected all the samples where their dissimilarity score was between [0.18, 0.32]. This decreases the size of potential desktop locker ransomware that were not detected by UNVEIL to 4,642 samples. We manually checked the post-attack screenshots of these samples, and found 377 desktop locker ransomware that UNVEIL was not able to detect. Our analysis shows that the false negatives in desktop locker ransomware resulted from samples in one ransomware family that generated a very transparent ransom note with a dissimilarity score between [0.27, 0.31] that was difficult to read.

For file locker ransomware, we first removed the samples that were not detected as malware by any of the AV scanners in VirusTotal after multiple resubmissions in consecutive days (see Section 5.3.2). By applying this approach, we were able to reduce the number of samples to check by 47%. Then, we applied a similar approach we used as described above. We listed the first five user files generated for that sample run and checked whether any process requested write access to those files. We also checked the entropy of multiple data buffers. If we identified write access with a significant increase in the entropy of data buffers compared to the entropy of data buffer in the read access for those files, we report it as a false negative.

Our test shows that UNVEIL does not have any false negatives in file locker ransomware samples. Consequently, we conclude that UNVEIL is able to detect multiple classes of ransomware attacks with a low false positive rate (FPs = 0.0% at a TP = 96.3%).

5.3.2 Early Warning

One of the design goals of UNVEIL is to be able to automatically detect previously unknown (i.e., zero-day) ransomware. In order to run this experiment, we did the following. Once per day over the course of the experiment, we built a malware dataset that was concurrently submitted to UNVEIL and VirusTotal. If a sample was detected as ransomware by UNVEIL, we checked the VirusTotal (VT) detection results. In cases where a ransomware sample was not detected by any VT scanner, we reported it as a new detection.

In addition, we also measured the lag between a new detection by UNVEIL and a VT detection. To that end, we created a dataset from the newly detected samples submitted on days $\{1, 2, \dots, n-1, n\}$ and re-submitted these samples to see whether the detection results changed. We considered the result of all 55 VT scanners in this experiment. Since the number of scanners is relatively high, we defined a VT detection ratio ρ as the ratio of the total number of scanners that identified the sample as ransomware or malware to the total number of scanners checked by VT. ρ is therefore a value on the interval $[0, 1]$ where zero means that the sample was not detected by any of the 55 VT scanners, and 1 means that all scanners reported the sample as malware or ransomware. Since there is no standard labeling scheme for malware in the AV industry, a scanner can label a sample using a completely different name from another scanner. Consequently, to avoid biased results, we consider the labeling of a sample using *any* name as a successful detection.

In our experiment, we submitted the detected samples every day to see how the VT detection ratio ρ changes over time. The distribution of ρ for each submission is shown in Figure 4. Our analysis shows that ρ does not significantly change after a small number of subsequent submissions. For the first submission, 72.2% of the ransomware samples detected by UNVEIL were not detected by any of the 55 VT scanners. After a few submissions, ρ does not change significantly, but generally was concentrated either towards small or very large ratios. This means that after a few re-submissions, either only a few scanners detected a sample, or almost all the scanners detected the sample.

5.4 Case Study: Automated Detection of a New Ransomware Family

In this section, we describe a new ransomware family, called SilentCrypt, that was detected by UNVEIL during the experiments. After our system detected these samples and submitted them to VirusTotal, several AV vendors picked up on them and also started detecting them a

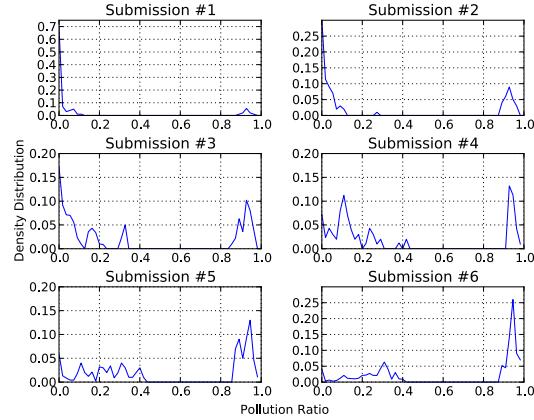


Figure 4: Evolution of VT scanner reports after six submissions. 72.2% of the samples detected by UNVEIL were not detected by any of AV scanners in the first submission. After a few re-submissions, the detection results do not change significantly. The detection results tend to be concentrated either towards small or very large detection ratios. This means that a sample is either detected by a relatively small number of scanners, or almost all of the scanners.

couple of days later, confirming the malice of the sample that we automatically detected.

This family uses a unique and effective method to fingerprint the runtime environment of the analysis system. Unlike other malware samples that check for specific artifacts such as registry keys, background processes, or platform-specific characteristics, this family checks the private files of a user to determine if the code is running in an analysis environment. When the sample is executed, it first checks the number of files in the user's directories, and sends this list to the C&C server before starting the attack.

Multiple online malware analysis systems such as malwr.com, Anubis, and a modern sandboxing technology provided by a well-known, anti-malware company did not register any malicious activity for this sample. However, the sample showed heavy encryption activity when analyzed by UNVEIL.

An analysis of the I/O activity of this sample revealed that this family first waited for several minutes before attacking the victim's files. Figure 5 shows the three main I/O activities of one of the samples in this family. The sample traverses the current user's main directories, and creates a list of files and folders. If the sample receives permission to attack from the C&C server, it begins encrypting the targeted files. To confirm UNVEIL's alerts, we conducted a manual investigation over several days. Our analysis concluded that the malicious activity is started only if user activity is detected. Unlike other ransomware samples that imme-

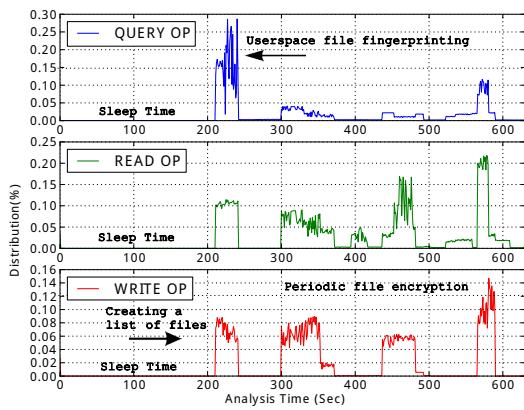


Figure 5: I/O activities of a previously unknown ransomware family detected by UNVEIL. The sample first performs victim file fingerprinting to ensure that the running environment is not a bare user environment.

diate attack a victim’s files when they are executed, this family only encrypt files that have recently been opened by the user while the malicious process is monitoring the environment. That is, the malicious process reads the file’s data and overwrites it with encrypted data if the file is used. The file name is then updated to “filename.extension.locked_forever” after it has been encrypted.

UNVEIL was able to detect this family of ransomware automatically because it was triggered after the system accessed some of the generated user files as a part of the user activity emulation scripts. Once we submitted the sample to VirusTotal, the sample was picked up by other AV vendors (5/55) after five days with different labels. A well-known, sandboxing-based security company confirmed our findings that the malware sample was a new threat that they had not detected before. We provide an anonymous video of a sample from this ransomware family in [6].

6 Discussion and Limitations

The evaluation in Section 5 demonstrates that UNVEIL achieves good, practical, and useful detection results on a large, real-world dataset. Unfortunately, malware authors continuously observe defensive advances and adapt their attacks accordingly. In the following, we discuss limitations of UNVEIL and potential evasion strategies.

There is always the possibility that attackers will find ways to fingerprint the automatically generated user environment and avoid it. However, this comes at a high cost, and increases the difficulty bar for the attacker. For example, in desktop-locking ransomware, malware can

use heuristics to look for specific user interaction before locking the desktop (e.g., waiting for multiple login events or counting the number of user clicks). However, implementing these approaches can potentially make detection easier since these approaches require hooking specific functions in the operating system. The presence of these hooking behaviors are themselves suspicious and are used by current malware analysis systems to detect different classes of malware. Furthermore, these approaches delay launching the attack which increases the risk of being detected by AV scanners on clients before a successful attack occurs.

Another possibility is that a malware might only encrypt a specific part of a file instead of aggressively encrypting the entire file, or simply shuffle the file content using a specific pattern that makes the files unreadable. Although we have not seen any sample with these behaviors, developing such ransomware is quite possible. The key idea is that in order to perform such activities, the malicious program should open the file with write permission and manipulate at least some data buffers of the file content. In any case, if the malicious program accesses the files, UNVEIL will still see this activity. There is no real reason for benign software to touch automatically generated files with write permission and modify the content. Consequently, such activities will still be logged. Malware authors might use other techniques to notify the victim and also evade the desktop lock monitor. As an example, the ransomware may display the ransom note via video or audio files rather than locking the desktop. As we partially discussed, these approaches only make sense if the malware is able to successfully encrypt user files first. In this case, UNVEIL can identify those malicious filesystem access as discussed in Section 3.1.2.

We also believe that the current implementation of text extraction to detect desktop locker ransomware can be improved. We observed that the change in the structure of the desktop screen-shots is enough to detect a large number of current ransomware attacks since UNVEIL exploits the attacker’s goal which is to ensure that the victims see the ransom note. However, we believe that the text extraction module can be improved to detect possible evasion techniques an attacker could use to generate the ransom note (e.g., using uncommon words in the ransom text).

Clearly, there is always the possibility that an attacker will be able to fingerprint the dynamic analysis environment. For example, stalling code [26] has become increasingly popular to prevent the dynamic analysis of a sample. Such code takes longer to execute in a virtual environment, preventing execution from completing during an analysis. Also, attackers can actively look for signs of dynamic analysis (e.g., signs of execution in a VM

such as well-known hard disk names). Note that UNVEIL is agnostic as to the underlying dynamic analysis environment. Hence, as a mitigation, UNVEIL can use a sandbox that is more resistant to these evasion techniques(e.g., [26, 48]). The main contribution of UNVEIL is not the dynamic analysis of malware, but rather the introduction of new techniques for the automated, specific detection of ransomware during dynamic analysis.

UNVEIL runs within the kernel, and aims to detect user-level ransomware. As a result, there is the risk that ransomware may run at the kernel level and thwart some of the hooks UNVEIL uses to monitor the filesystem. However, this would require the ransomware to run with administrator privileges to load kernel code or exploit a kernel vulnerability. Currently, most ransomware runs as user-level programs because this is sufficient to carry out ransomware attacks. Kernel-level attacks would require more sophistication, and would increase the difficulty bar for the attackers. Also, if additional resilience is required, the kernel component of UNVEIL could be moved outside of the analysis sandbox.

7 Related Work

Many approaches have been proposed to date that have aimed to improve the analysis and detection of malware. A number of approaches have been proposed to describe program behavior from analyzing byte patterns [29, 43, 41, 50] to transparently running programs in malware analysis systems [4, 23, 22, 47]. Early steps to analyze and capture the main intent of a program focused on analysis of control flow. For example, Kruegel et al. [28] and Bruschi et al. [9] showed that by modeling programs based on their instruction-level control flow, it is possible to bypass some forms of obfuscation. Similarly, Christodorescu et al. [12] used instruction-level control flow to design obfuscation-resilient detection systems. Later work focused on analyzing and detecting malware using higher-level semantic characterizations of their runtime behavior derived from sequences of system call invocations and OS resource accesses [24, 25, 11, 33, 42, 51].

Similar to our use of automatically-generated user content, decoys have been used in the past to detect security breaches. For instance, the use of decoy resources has been proposed to detect insider attacks [8, 52]. Recently, Juels et al. [18] used *honeywords* to improve the security of hashed passwords. The authors show that decoys can improve the security of hashed passwords since the attempt to use the decoy password for logins results in an alarm. In other work, Nikiforakis et al. [35] used decoy files to detect illegally obtained data from file hosting services.

There have also been some recent reports on the ransomware threat. For example, security vendors have reported on the threat of potential of ransomware attacks based on the number of infections that they have observed [46, 7, 44, 36]. A first report on specific ransomware families was made by Gazer where the author analyzed three ransomware families including Krotten and Gpcode [14]. The author concluded that while these early families were designed for massive propagation, they did not fulfill the basic requirements for mass extortion (e.g., sufficiently long encryption keys). Recently, Kharraz et al. [21] analyzed 15 ransomware families and provided an evolution-based study of ransomware attacks. They performed an analysis of charging methods and the use of Bitcoin for monetization. They proposed several high-level mitigation strategies such as the use of decoy resources to detect suspicious file access. Their assumption is that every filesystem access to delete or encrypt decoy resources is malicious and should be reported. However, they did not implement any concrete solution to detect or defend against these attacks.

We are not aware of any systems that have been proposed in the literature that specifically aim to detect ransomware in the wild. In particular, in contrast to existing work on generic malware detection, UNVEIL detects behavior specific to ransomware (e.g., desktop locking, patterns of filesystem accesses).

8 Conclusions

In this paper we presented UNVEIL, a novel approach to detecting and analyzing ransomware. Our system is the first in the literature to specifically identify typical behavior of ransomware such as malicious encryption of files and locking of user desktops. These are behaviors that are difficult for ransomware to hide or change.

The evaluation of UNVEIL shows that our approach was able to correctly detect 13,637 ransomware samples from multiple families in a real-world data feed with zero false positives. In fact, UNVEIL outperformed all existing AV scanners and a modern industrial sandboxing technology in detecting both superficial and technically sophisticated ransomware attacks. Among our findings was also a new ransomware family that no security company had previously detected before we submitted it to VirusTotal.

9 Acknowledgements

This work was supported by the National Science Foundation (NSF) under grant CNS-1409738, and Secure Business Austria.

References

- [1] Minotaur Analysis - Malware Repository. minotauranalysis.com.
- [2] Malware Tips - Your Security Advisor. <http://malwaretips.com/forums/viruses-exchange.104/>.
- [3] MalwareBlackList - Online Repository of Malicious URLs. <http://www.malwareblacklist.com>.
- [4] Proof-of-concept Automated Baremetal Malware Analysis Framework. <https://code.google.com/p/nvctrace/>.
- [5] BitBlaze Malware Analysis Service. <http://bitblaze.cs.berkeley.edu/>, 2016.
- [6] SilentCrypt: A new ransomware family. <https://www.youtube.com/watch?v=qIASKA4BMck>, 2016.
- [7] AJAN, A. Ransomware: Next-Generation Fake Antivirus. http://www.sophos.com/en-us/media/library/PDFs/technicalpapers/Sophos_RansomwareFakeAntivirus.pdf, 2013.
- [8] BOWEN, B. M., HERSHKOP, S., KEROMYTIS, A. D., AND STOLFO, S. J. *Baiting inside attackers using decoy documents*. Springer, 2009.
- [9] BRUSCHI, D., MARTIGNONI, L., AND MONGA, M. Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment*. Springer, 2006, pp. 129–143.
- [10] CATALIN CIMPANU. Breaking Bad Ransomware Completely Undetected by VirusTotal. <http://http://news.softpedia.com/news/breaking-bad-ransomware-goes-completely-undetected-by-virustotal-493265.shtml>, 2015.
- [11] CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference* (2008), ACM, pp. 5–14.
- [12] CHRISTODORESCU, M., JHA, S., SESIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on* (2005), IEEE, pp. 32–46.
- [13] CUCKOO FOUNDATION. Cuckoo Sandbox: Automated Malware Analysis. www.cuckoosandbox.org, 2015.
- [14] GAZET, A. Comparative analysis of various ransomware virii. *Journal in Computer Virology* 6, 1 (February 2010), 77–90.
- [15] GRIER, C., BALLARD, L., CABALLERO, J., CHACHRA, N., DIETRICH, C. J., LEVCHENKO, K., MAVROMMATHIS, P., MCCOY, D., NAPPA, A., PITSILLIDIS, A., ET AL. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 821–832.
- [16] INTERNATIONAL SECURE SYSTEM LAB. Anubis - Malware Analysis for Unknown Binaries. <https://anubis.iseclab.org/>, 2015.
- [17] JASHUA TULLY. An Anti-Reverse Engineering Guide. <http://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide#StolenBytes>, 2008.
- [18] JUELS, A., AND RIVEST, R. L. Honeywords: Making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 145–160.
- [19] KAWAKOYA, Y., IWAMURA, M., SHIOJI, E., AND HARIU, T. Api chaser: Anti-analysis resistant malware analyzer. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2013, pp. 123–143.
- [20] KEVIN SAVAGE, PETER COOGAN, HON LAU. the Evolution of Ransomware. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-evolution-of-ransomware.pdf, 2015.
- [21] KHARRAZ, A., ROBERTSON, W., BALZAROTTI, D., BILGE, L., AND KIRDA, E. Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (07 2015).
- [22] KIRAT, D., VIGNA, G., AND KRUEGEL, C. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 403–412.
- [23] KIRAT, D., VIGNA, G., AND KRUEGEL, C. Barecloud: Bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), USENIX Association, pp. 287–301.

- [24] KIRDA, E., KRUEGEL, C., BANKS, G., VIGNA, G., AND KEMMERER, R. Behavior-based spyware detection. In *Usenix Security* (2006), vol. 6.
- [25] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X.-Y., AND WANG, X. Effective and efficient malware detection at the end host. In *USENIX security symposium* (2009), pp. 351–366.
- [26] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 285–296.
- [27] KREBS, B. FBI: North Korea to Blame for Sony Hack. <http://krebsonsecurity.com/2014/12/fbi-north-korea-to-blame-for-sony-hack/>, 2014.
- [28] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection* (2006), Springer, pp. 207–226.
- [29] LI, W.-J., WANG, K., STOLFO, S. J., AND HERZOG, B. Fileprints: Identifying file types by n-gram analysis. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC* (2005), IEEE, pp. 64–71.
- [30] LIN, J. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory* 37 (1991), 145–151.
- [31] LINDORFER, M., KOLBITSCH, C., AND COMPARETTI, P. M. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection* (2011), Springer, pp. 338–357.
- [32] MALWARE DON'T NEED COFFEE. Guess who's back again ? Cryptowall 3.0. <http://malware.dontneedcoffee.com/2015/01/guess-whos-back-again-cryptowall-30.html>, 2015.
- [33] MARTIGNONI, L., STINSON, E., FREDRIKSON, M., JHA, S., AND MITCHELL, J. C. A layered architecture for detecting malicious behaviors. In *Recent Advances in Intrusion Detection* (2008), Springer, pp. 78–97.
- [34] MICROSOFT, INC. File System Minifilter Drivers. <https://msdn.microsoft.com/en-us/library/windows/hardware/ff540402%28v=vs.85%29.aspx>, 2014.
- [35] NIKIFORAKIS, N., BALDUZZI, M., ACKER, S. V., JOOSEN, W., AND BALZAROTTI, D. Exposing the lack of privacy in file hosting services. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats (LEET)* (March 2011), LEET 11, USENIX Association.
- [36] O'GORMAN, G., AND McDONALD, G. Ransomware: A Growing Menace. <http://www.symantec.com/connect/blogs/ransomware-growing-menace>, 2012.
- [37] PAYLOAD SECURITY INC,. Payload Security. <https://www.hybrid-analysis.com>, 2016.
- [38] RAY SMITH. Tesseract Open Source OCR Engine . <https://github.com/tesseract-ocr/tesseract>, 2015.
- [39] REAQTA INC,. HyraCrypt Ransomware. <https://reaqta.com/2016/02/hydracrypt-ransomware/>, 2016.
- [40] ROSSOW, C., DIETRICH, C. J., GRIER, C., KREIBICH, C., PAXSON, V., POHLMANN, N., BOS, H., AND VAN STEEN, M. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 65–79.
- [41] SCHULTZ, M. G., ESKIN, E., ZADOK, E., AND STOLFO, S. J. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on* (2001), IEEE, pp. 38–49.
- [42] STINSON, E., AND MITCHELL, J. C. Characterizing bots remote control behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2007, pp. 89–108.
- [43] SUNG, A. H., XU, J., CHAVEZ, P., AND MUKKAMALA, S. Static analyzer of vicious executables (save). In *Computer Security Applications Conference, 2004. 20th Annual* (2004), IEEE, pp. 326–334.
- [44] SYMANTEC, INC. Internet Security Threat Report. http://www.symantec.com/security_response/publications/threatreport.jsp, 2014.
- [45] THE CYBER THREAT ALLIANCE. Lucrative Ransomware Attacks: Analysis of Cryptowall Version 3 Threat. <http://cyberthreatalliance.org/cryptowall-report.pdf>, 2015.
- [46] TRENDLABS. An Onslaught of Online Banking Malware and Ransomware. <http://apac.trend>

- micro.com/cloud-content/apac/pdfs/security-intelligence/reports/rpt-cashing-in-on-digital-information.pdf, 2013.
- [47] VASUDEVAN, A., AND YERRABALLI, R. Co-bra: Fine-grained malware analysis using stealth localized-executions. In *Security and Privacy, 2006 IEEE Symposium on* (2006).
- [48] VIGNA, G. From Anubis and Wepawet to Llama. <http://info.lastline.com/blog/from-anubis-and-wepawet-to-llama>, June 2014.
- [49] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on* 13, 4 (2004), 600–612.
- [50] XU, J.-Y., SUNG, A. H., CHAVEZ, P., AND MUKKAMALA, S. Polymorphic malicious executable scanner by api sequence analysis. In *Hybrid Intelligent Systems, 2004. HIS'04. Fourth International Conference on* (2004), IEEE, pp. 378–383.
- [51] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 116–127.
- [52] YUILL, J., ZAPPE, M., DENNING, D., AND FEER, F. Honeyfiles: deceptive files for intrusion detection. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC* (2004), IEEE, pp. 116–122.

A Benign Applications Used in Experiment One

Application	Main Capability	Version
7-zip	Compression	15.06
Winzip	Compression	19.5
WinRAR	Compression	5.21
DiskCryptor	Encryption	1.1.846.118
AESCrypt	Encryption	—
Eraser	Shredder	6.2.0.2969
SDelete	Shredder	1.61

Table 5: The list of benign applications that generate similar I/O access patterns to ransomware.

Towards Measuring and Mitigating Social Engineering Software Download Attacks

Terry Nelms^{1,2}, Roberto Perdisci^{3,1}, Manos Antonakakis¹, and Mustaque Ahamed^{1,4}

¹Georgia Institute of Technology

²Damballa, Inc.

³University of Georgia

⁴New York University Abu Dhabi

tnelms@gatech.edu, perdisci@cs.uga.edu, manos@gatech.edu, mustaq@cc.gatech.edu

Abstract

Most modern malware infections happen through the browser, typically as the result of a drive-by or social engineering attack. While there have been numerous studies on measuring and defending against drive-by downloads, little attention has been dedicated to studying social engineering attacks.

In this paper, we present the first systematic study of web-based social engineering (SE) attacks that successfully lure users into downloading malicious and unwanted software. To conduct this study, we collect and reconstruct more than two thousand examples of in-the-wild SE download attacks from live network traffic. Via a detailed analysis of these attacks, we attain the following results: (i) we develop a categorization system to identify and organize the tactics typically employed by attackers to gain the user’s attention and deceive or persuade them into downloading malicious and unwanted applications; (ii) we reconstruct the web path followed by the victims and observe that a large fraction of SE download attacks are delivered via online advertisement, typically served from “low tier” ad networks; (iii) we measure the characteristics of the network infrastructure used to deliver such attacks and uncover a number of features that can be leveraged to distinguish between SE and benign (or non-SE) software downloads.

1 Introduction

Most modern malware infections happen via the browser, typically triggered by social engineering [9] or drive-by download attacks [33]. While numerous studies have focused on measuring and defending against drive-by downloads [14, 17, 28, 38], malware infections enabled by social engineering attacks remain notably understudied [31].

Moreover, as recent defenses against drive-by downloads and other browser-based attacks are becoming harder to circumvent [18, 24, 32, 36, 40], cyber-criminals increasingly aim their attacks against the weakest link,

namely the user, by leveraging sophisticated social engineering tactics [27]. Because social engineering (SE) attacks target users, rather than systems, current defense solutions are often unable to accurately detect them. Thus, there is a pressing need for a comprehensive study of social engineering downloads that can shed light on the tactics used in modern attacks. This is important not only to inform better technical defenses, but may also allow us to gather precious information that may be used to better train users against future SE attacks.

In this paper, we present a study of real-world SE download attacks. Specifically, we focus on studying *web-based SE attacks* that unfold *exclusively via the web* and that do not require “external” triggers such as email spam/phishing, etc. An example of such attacks is described in [9]: a user is simply browsing the web, visiting an apparently innocuous blog, when his attention is drawn to an online ad that is subtly crafted to mimic a warning about a missing browser plugin. Clicking on the ad takes him to a page that reports a missing codec, which is required to watch a video. The user clicks on the related codec link, which results in the download of malicious software.

To conduct our study, we collect and analyze hundreds of *successful* in-the-wild SE download attacks, namely SE attacks that actually result in a victim downloading malicious or unwanted software. We harvest these attacks by monitoring *live* web traffic on a large academic network. Via a detailed analysis of our SE attack dataset, we attain the following main results: (i) we develop a categorization system to identify and organize the tactics typically employed by attackers to gain a user’s attention and deceive or persuade them into downloading malicious and unwanted applications; (ii) we reconstruct the *web path* (i.e., sequence of pages/URLs) followed by SE victims and observe that a large fraction of SE attacks are delivered via online advertisement (e.g., served via “low tier” ad networks); (iii) we measure the characteristics of the network infrastructure (e.g., domain names) used to

deliver such attacks, and uncover a number of features that can be leveraged to distinguish between SE and benign (i.e., “non-SE”) software downloads.

Our findings show that a large fraction of SE attacks (almost 50%) are accomplished by *repackaging* existing benign applications. For instance, users often download free software that comes as a bundle including the software actually desired by the user plus some Adware or other Potentially Unwanted Programs (PUPs). This confirms that websites serving free software are often involved (willingly or not) in distributing malicious or unwanted software [4, 7].

The second most popular category of attacks is related to alerting or urging the user to install an application that is supposedly needed to complete a task. For instance, the user may be *warned* that they are running an outdated or insecure version of Adobe Flash or Java, and are offered to download a software update. Unfortunately, by downloading these supposed updates, users are infected. Similarly, users may stumble upon a page that supposedly hosts a video of interest. This page may then inform the user that a specific video codec is needed to play the desired video. The user *complies* by downloading the suggested software, thus causing an infection (see Section 3 for details).

Another example of an SE download attack is represented by fake anti-viruses (FakeAVs) [35]. In this case, a web page alerts the user that their machine is infected and that AV software is needed to clean up the machine. In a way similar to the SE attack examples reported above, the user may be persuaded to download (in some cases after a payment) the promoted software, which will infect the user’s machine. However, while FakeAVs have been highly popular among attackers in the recent past, our study of in-the-wild SE malware downloads finds that they represent less than 1% of modern SE attacks. This sharp decline in the number of FakeAV attacks within the last few years is consistent with the recent development of technical countermeasures against this class of attacks [5] and increased user awareness [6].

As mentioned earlier, a large fraction of SE download attacks (more than 80%) are initiated via advertisements, and that the “entry point” to these attacks is represented by only a few low-tier advertisement networks. For instance, we found that a large fraction of the web-based SE attacks described above are served primarily via two ad networks: `oncickads.net` and `adcash.com`.

By studying the details of SE download attacks, we also discover a number of features that aid in the detection of SE download attacks on live web traffic. We train a classifier using these features and measure its effectiveness at detecting SE downloads.

Summary of Contributions:

- We present the first systematic study of modern web-based SE download attacks. For instance, our analysis of hundreds of SE download attack instances reveals that most such attacks are enabled by online advertisements served through a handful of “low tier” ad networks.
- To assist the process of understanding the origin of SE download attacks, we develop a categorization system that expresses how attackers typically gain a user’s attention, and what are the most common types of deception and persuasion tactics used to trick victims into downloading malicious or unwanted applications. This makes it easier to track what type of attacks are most prevalent and may help to focus user training programs on specific user weaknesses and particularly successful deception and persuasion tactics currently used in the wild.
- Via extensive measurements, we find that the most common types of SE download attacks include *fake updates* for Adobe Flash and Java, and that fake anti-viruses (FakeAVs), which have been a popular and effective infection vector in the recent past, represent less than 1% of all SE downloads we observed in the wild. Furthermore, we find that existing defenses, such as traditional anti-virus (AV) scanners, are largely ineffective against SE downloads.
- Based on our measurements, we then identify a set of features that allow for building a statistical classifier that is able to accurately detect ad-driven SE download attacks with 91% true positives and only 0.5% false positives.

2 Study Overview

Our study of SE download attacks is divided in multiple parts. To better follow the results discussed in the following sections, we now present a brief summary of their content.

In Section 3, we analyze the range of deception and persuasion tactics employed by the attackers to victimize users, and propose a categorization system to systematize the in-the-wild SE tactics we observed.

In Section 4, we discuss how we collect software downloads (including malicious ones) from live network traffic and reconstruct their *download path*. Namely, we trace back the sequence of pages/URLs visited by a user before arriving to a URL that triggers the download of an executable file (we focus on Windows portable executable files). We then analyze the collected software download events, and label those downloads that result

from SE attacks. This labeled dataset is used in the following sections to enable a detailed analysis of the characteristics of the SE download attacks.

We analyze our dataset of in-the-wild SE download attacks in Section 5. Specifically, we measure how the SE attack tactics are distributed, according to the categorization system proposed in Section 3, and highlight the most popular *successful* SE malware attacks. According to our dataset, the majority of SE attacks are promoted via online advertisement. Therefore, in Section 5 we also present an analysis of the network-level properties of ad-based SE malware attacks, and contrast them with properties of ad-driven benign software downloads.

In Section 6, we focus on detecting ad-based SE download attacks. We first show that anti-virus products detect only a small fraction of all SE attacks, leaving most “fresh” SE download events undetected. We then devise a number of statistical features that can be extracted from the network properties of ad-driven software download events, and show that they allow us to build an accurate SE attack classifier, which could be used to detect and stop SE download attacks before they infect their victims.

Finally, we discuss possible limitations of our SE attacks study and detection approach in Sections 7, and contrast our work to previously published research in Section 8.

3 SE Download Attacks

In this section, we analyze the range of deception and persuasion tactics employed by the attackers to victimize users (Section 3.1). We also provide some concrete examples of SE download attacks, to highlight how real users may fall victim to such attacks (Section 3.2).

SE Attacks Dataset. Our analysis is based on a dataset consisting of 2,004 real-world SE download attacks. We collected these attacks by monitoring the network traffic of a large academic network (authorized by our organization’s IRB), passively reconstructing the download of executable binary files and tracing back the browsing path followed by the users to reach the file download event. We then analyzed the observed file download events to identify possible malware, adware or PUP downloads. Finally, we performed an extensive manual analysis of the suspicious downloads to identify and label those downloads that were triggered by SE attacks, and to precisely reconstruct the attack scenarios. A detailed description of our dataset collection and labeling approach is provided in Section 4. Furthermore, in Section 5 we measure properties of the collected attacks, such as what types of SE attacks are the most prevalent, and provide information on the network-level characteristics of SE download distribution operations.

In the following, we will focus on analyzing our SE download attack dataset with the goal of categorizing the different types of deception and persuasion tactics used by attackers to lure victims into downloading malicious and unwanted software.

3.1 Categorizing SE Download Tactics

The dataset of 2,004 SE download attacks that we reconstructed and labeled via extensive manual analysis efforts (detailed in Section 4) gives us an excellent opportunity to study the wide range of depiction and persuasion tactics employed by the attackers. To better understand how SE attacks work, we develop a categorization system that aims to provide a systematization of the techniques used by successful SE download attacks. Specifically, we categorize different SE attacks according to; (1) the ways the adversaries get the user’s attention and (2) the type of deception and persuasion tactics employed. Our categorization of SE attacks is summarized in Figure 1.

Gaining the user’s attention. The first step in a SE attack is to get the user’s attention. This is accomplished for example by leveraging online advertisement (ad), search engine optimization (SEO) techniques or by posting messages (and clickable links) on social networks, forums, and other sites that publish user-generated content.

As we will show in Section 5, the most popular of these methods is *ads*. On-line advertisement allows the attacker to easily “publish” their deception/persuasion ad on a site that is likely already popular among the targeted victims. In addition, ads help hide the deception/persuasion campaign and attack infrastructure (i.e., hide it from users as well as security researchers), simply because SE ads are exposed only to targeted users via search keywords, the user’s cookies, etc.

Another method employed to attract the user’s attention is *search*. For instance, search engines can be abused via black hat SEO attacks to pollute the search results with harmful links. In addition, in our categorization of SE attacks we use a generic definition of “search” that does not only include search engines; anytime a user perform a query to locate specific content on a website, we classify it as a search event. For instance, we have observed users that become victims of SE attacks while simply searching for content within a website that hosts video streaming (e.g., movies, video clips, etc.).

Attackers also use *web posts* to attract the user’s attention. We define a web post as content that has been added to a website by a visitor and is now available for display to others. For instance, many of the web posts used in the SE download attacks we observed were located within groups of legitimate posts about a topic of interest. The majority of such web posts were related to content (e.g.,

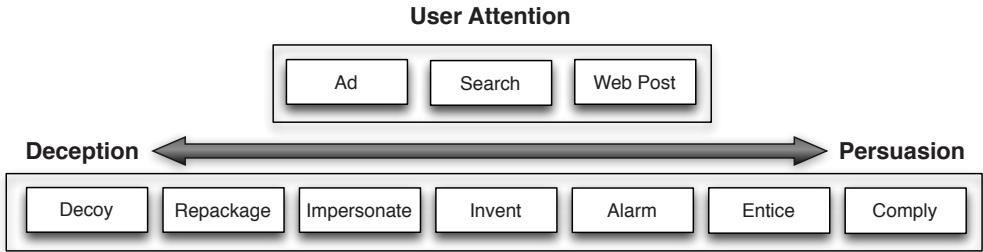


Figure 1: Categorization of SE downloads on the web.

clickable links) such as free software, books, music and movies.

It is not uncommon for these three techniques (ads, search, and web posts) to be combined. For instance, attackers will use search and ads in combination to get the user’s attention. Targeted search engine ads related to the search terms entered by users are often displayed before the real search results, thus increasing the likelihood of a click. This common search engine feature is often abused by attackers. Also, users may search for certain specific terms on web forums, social network sites, etc., and may fall victim to targeted web posts.

Deception and persuasion tactics. After an attacker gains the user’s attention, they must convince them to download and install their malicious or unwanted software. This typically involves combining a subset of the deception and persuasion techniques summarized in Figure 1. As one scrolls from left to right in the figure, the techniques move from deception towards persuasion. Notice that none of the techniques we list involve only deception or only persuasion; instead, the different techniques vary in their levels of each. We now provide a description of the deception and persuasion classes shown in Figure 1. We will then present examples of real-world SE download attacks that make use of a combination of these techniques.

- (1) **Decoy:** Attackers will purposely place decoy “clickable” objects, such as a hyperlink, at a location on a web page that will attract users to it and away from the actual object desired (or searched) by the user. An image of a “flashy” download button (e.g., delivered as an ad banner) on a free download site located prior to the actual download link desired by the user is an example of this technique.
- (2) **Repackage:** To distribute malicious and unwanted software, attackers may group benign and PUP (or malware) executables together, and present them to the user as a single application. An example technique from this class is adware bundled with a benign application and served as a single software download on a free software distribution website.
- (3) **Impersonate:** Using specific images, words and colors can make an executable appear as if it was a

known popular benign application. Also, claiming that a software provides desirable features or services (though it does not supply them) is a way to convince the user to download and install the application. Malicious executables pretending to be an Adobe Flash Player update, e.g., by using logos or icons similar to the original Adobe products and keywords such as “adobe” and “flash,” is an example of impersonation techniques from this class.

- (4) **Invent:** Creating a false reality for the user may compel them to download a malicious or unwanted executable. For example, alerting the user stating that their machine is infected with malware and instructing them to download (malicious) clean-up software (e.g., a fake AV) is an example of the *invent* tactic.
- (5) **Alarm:** Using fear and trepidation aims to scare the user into downloading (malicious) software that promises to safeguard them. For instance, an online ad claiming that the user’s browser is out-of-date and vulnerable to exploitation is an example of *alarm* techniques.
- (6) **Entice:** Attackers often attempt to attract users to download a malicious or unwanted executable by offering features, content or advantages. As an example, a user may be shown an ad for a system optimization utility stating that it will “speedup” their PC, but hides malicious software.
- (7) **Comply:** A user may be (apparently) required to install an (malicious) application before she can continue. For instance, a user visiting a video streaming website may be prompted to install a necessary “codec” before she can watch a free movie. As the user is motivated to watch the movie, she complies with the codec installation request, thus getting infected with malware.

It is important to note that none of the SE attacks in our study fall into a single class. Instead the in-the-wild SE attacks we collected often use techniques across two or more of the above classes to trick the user into infecting their machine. Labeling a download using these classes involves understanding the motivations employed to convince a user to install the malicious software. These are

typically easy to identify by examining the words and images used in an attack. For instance, an attack that *impersonates* will claim to be software that it is not, such as Adobe Flash Player. On the other hand, an attack that *enticing* a user will often use words like “free” and describe all the benefits of installing the software. Entice and impersonate are not mutually exclusive and are used together in some SE attacks. Allowing an SE attack to be assigned to more than one class simplifies the labeling process because all perception/deception tactics can be included, not just the one believed to be the primary tactic.

3.2 Examples of In-The-Wild SE Attacks

In this section, we present two examples from our dataset of reconstructed SE download attacks, and classify their SE tactics using our categorization system (see Figure 1). To aid in our discussion we define the notation “*attention:deception/persuasion*,” where the *attention* string refers to how attackers attempt to attract users’ attention, and the *deception/persuasion* string refers to the combination of the deception/persuasion techniques used to trigger the malware download. For example, if an SE attack relies on an *ad* and uses *alarm* and *impersonate* deception/persuasion tactics, then we label the attack using our notation as “*ad:alarm+impersonate*.”

Attack 1. A user searches for “Gary Roberts free pics” using a popular search engine. A page hosted on a compromised website is returned as a top result. The page contains various content referring to “Gary Roberts”, but this content is incoherent and likely only present for blackhat search engine optimization (SEO). However, the user never sees the content because the javascript code located at the top of the served page immediately closes the document, and then reopens it to inject a script that redirects the user to a page that says “gary-roberts-free-pics is ready for download. Your file download should automatically start within seconds. If it doesn’t, click to restart the download.” But the downloaded file does not contain any pictures, and instead carries malicious code that is later flagged as malware by some AV vendors.

Using our categorization system we classify this attack as “*search:entice+decoy+impersonate*.” *Search* is the method of gaining the users attention. In this example this is obvious because the SE page appeared in the results page provided by a search engine. The *entice* part of the attack is the offering of “free” pics of the subject of interest. *Decoy* is due to the fact that blackhat SEO was used to elevate the SE page in the search results above other legitimate pages. Lastly, what the user downloads is not pics of Gary Roberts; instead, it is a malicious executable *impersonating* what the user wants (e.g., Gary



Figure 2: SE ad for Ebola early warning system.

Roberts free pictures).

Attack 2. A user is watching an episode of “Agents of Shield” on a free video website when they are presented with an *ad*. The ad, similar to the one shown in Figure 2, presents the user with the option of downloading an early warning system for Ebola. However, the downloaded file does not provide information about an Ebola outbreak; instead, it infects the user’s system with malicious software.

We classify this attack as “*ad:alarm+impersonate*” using our categorization system. The user’s attention is gained through an *ad*, in which their fear of Ebola is used to *alarm* the user into downloading a tracking system. Unfortunately, what the user downloads only *impersonates* a tracking systems, and instead contains malicious code.

4 Collecting and Labeling SE Attacks

In this section we discuss in detail how we collected and labeled our dataset of 2,004 SE download attacks. We will then present an analysis of the prevalence and characteristics of the collected attacks in Section 5.

4.1 Data Collection Approach

To collect and reconstruct SE download attacks, we monitor all web traffic at the edge of a large network (this study was authorized by our organization’s Institutional Review Board). Using deep packet inspection, we process the network traffic in real-time, reconstructing TCP connections, analyzing the content of HTTP responses and recording all traffic related to the download of executable files (Windows executables).

While monitoring the traffic, we maintain a buffer of all recent HTTP transactions (i.e., request-response pairs) that occurred in the past few minutes. When an HTTP transaction that carries the download of an executable file is found, we passively reconstruct and store a copy of the file itself. In addition, we trigger a dump of the traffic buffer, recording all the web traffic generated

by the same client that initiated the file download during the past few minutes before the download started. In other words, we store all HTTP traffic a client generated up to (and including) the executable file download.

We then process these HTTP transaction captures using the trace-back algorithm presented in [30]. The trace-back algorithm builds a graph where each node is an HTTP transaction. Given two nodes T_1 and T_2 , they are connected by an edge if T_2 was “likely referred to” by T_1 . For instance, a directed edge is drawn from T_1 to T_2 if the user clicked on a link in T_1 ’s page and as a consequence the browser loaded T_2 . Then, starting from the download node, the algorithm walks backwards along this graph to trace back the most likely path (i.e., sequence of HTTP transactions) that brought the user to initiate the download event. For more details, we refer the reader to [30].

These reconstructed download paths are later analyzed to identify and categorize SE download attacks. It is important to note that we do not claim automatic download path traceback as a contribution of this paper. Instead, our focus is on the collection, analysis, and categorization of SE download attacks, and on the detection and mitigation of ad-based SE infections. Automatic download traceback is just one of the tools we use to aid in our analysis.

We deployed the data collection process described above on a large academic network serving tens of thousands users for a period of two months. To avoid unnecessarily storing the download traces related to frequent software updates for popular benign software, we compiled a conservative whitelist consisting of 128 domain names owned by major software vendors (e.g., Microsoft, Adobe, Google, etc.). Therefore, executable files downloaded from these domains were excluded from our dataset.

Overall, during our two month deployment, we collected a total of 35,638 executable downloads. The process we used to identify the downloads due to SE attacks is described in the following sections.

4.2 Automatic Data Filtering

Even though we filter out popular benign software updates up front, we found that the majority of executable downloads observed on a network are updates to (more or less popular) software already installed on systems. As we are interested in new infections caused by web-based SE attacks, we aim to automatically identify and filter out such software updates.

To this end, we developed a set of conservative heuristics that allow us to identify and filter out most software update events based on an analysis of their respective *download path*. First, we examine the *length* of the download path. The intuition is that software updates

tend to come from very short download paths, which often consist of a single HTTP request to directly fetch a new executable file from a software vendor’s website (or one of its mirrors). Conversely, the download path related to SE download attacks usually consists of a number of navigation steps (e.g., the user may navigate through different pages before stumbling upon a malicious SE advertisement).

For the next step in the analysis, we review the *user-agent* string observed in the HTTP requests on the download path. The user-agent string appearing in software update requests is typically not the one used by the client’s browser (similar observations were made by the authors of [30]), because the user-agent found in these requests often contains the name of the software that is being updated (e.g., Java or Acrobat reader). Since web-based SE attacks happen to users browsing the web, HTTP requests on the download path typically carry the user-agent string of the victim’s browser.

Therefore, to automatically identify and filter out update downloads we use the following heuristics. If the download path contains a single HTTP transaction (the update request itself), and the user-agent string does not indicate that the request has been made by a browser, we filter out the event from our dataset.

Overall, the conservative filtering approach outlined above allowed us to reduce the number of download paths to be further analyzed. Specifically, we were able to reduce our download traces dataset by 61%, leaving us with a total of 13,762 that required further analysis and labeling.

4.3 Analysis of Software Download Events

After filtering, our dataset consists of 13,762 software download events (i.e., the downloaded executable files and related download paths) that required further detailed analysis and labeling. As our primary goal is to create a high quality dataset of labeled SE download attacks, we aim to manually analyze and perform a detailed reconstruction of the attacks captured by our archive of software download events.

To aid in the manual analysis process and reduce the cost of this time-consuming effort, we leveraged unsupervised learning techniques. Specifically, we identify a number of statistical features that allow us to discover clusters of download events that are similar to each other. For instance, we aim to group different downloads of the same benign software by different clients. At the same time, we also aim to group together similar download events triggered by the same SE attack campaign.

To identify and automatically cluster similar download events, we developed a set of statistical features. We would like to emphasize that none of the features we describe below is able to independently yield high-quality

clustering results. Rather, it is the combination of these features that allows us to obtain high quality clusters of related software download events.

Notice also that the purpose of this clustering process is simply to reduce the time needed to manually analyze the software download events we collected. By using a conservative clustering threshold (discussed below) and by manually reviewing all obtained clusters, we minimize the impact of possible noise in the results.

To perform the clustering, we leverage a number of simple statistical features, some of which (e.g., URL similarity, domain name similarity, etc.) are commonly used to find the similarity between network-level events. Notice, however, that our main goal in this clustering process is not to design novel features; rather, we simply aim to reduce the manual analysis and labeling efforts needed to produce a high-quality dataset of in-the-wild SE download attacks.

We now describe our clustering features:

- (1) **Filename Similarity:** Benign executable files distributed by the same organization (e.g., an application distributed by a given vendor or software distribution site) tend to have similar filenames. Notice that often this also holds for SE attack campaigns, because the files distributed by the same campaign often follow a consistent “theme” to aid in the *deception* of the end users. For instance, the malware files distributed by a fake Flash Player upgrade attack campaign (see Section 3) may all include the word “Adobe” in the filename to convince the user that the downloaded file is legitimate.
- (2) **File Size Similarity:** Benign files that are identical or variants (i.e., different versions) of the same software are usually very close in size. Similarly, SE campaigns typically infect victims with a variant of the same malware family. While the malware file’s size may vary due to polymorphism, the size difference is typically small, compared to the total file size.
- (3) **URL Structure Similarity:** A benign website that serves software downloads will often host all of its executable files at the same or very similar structured URLs. In a similar way, SE campaigns often use malware distribution “kits” and go weeks or even months before a noticeable change in the structure of their URL paths is observed. This is unlike malicious URLs, which frequently change to avoid blacklisting.
- (4) **Domain Name Similarity:** While the domain names used to distributed malware files belonging to the same SE attack campaign may change, some campaigns will reuse some keywords in their domains that are meant to deceive the user. For instance, the domains used in a Fake AV malware campaign

may contain the keyword “security” to convince the user of its legitimacy. Also, download events related to (different versions of) the same benign software are often distributed via a handful of stable domain names.

- (5) **Shared Domain Predecessor:** SE attacks that share a common node (or predecessor) in the download path are often related. For instance, an SE malware campaign may exploit an ad network with weak anti-abuse practices. Therefore, while the final domain in the download path from which the malware is actually downloaded may change (e.g., to avoid blacklisting), the malware download *paths* of different users that fall victim to the same SE campaign may share a node related to the abused ad network, for example. On the other hand, in case of benign downloads both the download and “attention grabbing” domain tend to be stable, as the main goal is quality of service towards the end user.
- (6) **Shared Hosting:** While domains involved in malware distribution tend to change frequently, SE malware campaigns often reuse (parts of) the same hosting infrastructure (e.g., some IPs). The intuition is that hosting networks that tolerate abuse (knowingly or otherwise) are a rare and costly resource. On the other hand, domain names are significantly easier to obtain and can be used as crash-and-burn resource from the adversary. On the benign downloads side, legitimate software distribution websites are usually stable and do not change hosting very frequently, for quality of service reasons.
- (7) **HTTP Response Header Similarity:** The headers in an HTTP response are the result of the installed server-side software and configuration of the web server. The set of response headers and their associated values offer a lot of variation. However, most of the web servers for a benign site tend to have common configurations so they respond with similar headers. Also, some SE campaigns use the same platform for their attacks and do not change their server-side configurations even when they move to new domains.

For each of the 13,762 downloads we compute a feature vector based on the features listed above, and calculate the pairwise distance between these feature vectors. We then apply an agglomerative hierarchical clustering algorithm to the obtained distance matrix. Finally, we cut the dendrogram output by the hierarchical clustering algorithm to obtain the final clusters of download events. To cut the dendrogram we chose a conservative cut height to error on the side of not grouping related downloads and significantly reduce the possibility of grouping unrelated ones. This process produced 1,205 clusters, thus resulting in an order of magnitude reduc-

tion in the number of items that require manual inspection. In the following section we explain how we analyzed and labeled these clusters.

4.4 Labeling SE Download Attacks

After clustering similar software download events, we manually examine each cluster to distinguish between those that are related SE download attack campaigns, and clusters related to other types of software download events, including benign downloads, malware downloads triggered by drive-by downloads, and (benign or malicious) software updates. This labeling process allows us to focus our attention on studying SE download attacks, and to exclude other types of benign and malicious downloads (notice that because in this paper we are primarily interested on SE attacks, we exclude non-SE malware attacks from our study).

To perform the cluster labeling, each cluster was manually reviewed by randomly sampling 10% of the download events grouped in the cluster, and then performing a detailed manual analysis of the events in this sample set. For small clusters (e.g. clusters with < 50 events) we sampled a minimum of 5 download events. For clusters containing less than 5 download traces, we reviewed all of the events. As discussed earlier, our clustering process uses a conservative cut height. The net effect is that the clusters tend to be “pure,” thus greatly reducing the possibility of errors during the cluster labeling process. At the same time, some groups of download events that are similar to each other may be split into smaller clusters. However, this does not represent a significant problem for our labeling process. The only effect of having a larger number of highly compact clusters is to create additional manual work, since a random sample of events from each cluster is manually analyzed.

In addition to manually reviewing the download paths contained in the clusters, to assist our labeling we also make use of antivirus (AV) labels for the downloaded executable files. To increase AV detections we “aged” the downloads in our dataset for a period of two months, before scanning them with more than 40 AV engines using virustotal.com. Notice that AV labels are mainly used for confirmation purposes. The actual labeling of SE attacks is performed via an extensive review of each download path (i.e., sequence of pages/URLs visited to arrive to the executable file download). If we suspect a cluster is malicious (based on our manual analysis), having one or more AV hits offers additional confirmation, but is not required if we have strong evidence that the download was triggered by an SE attack.

Updates: Even though the heuristics we described in Section 4.2 filter out the vast majority of software updates, our heuristics are quite conservative and therefore some update events may still remain. To determine if

a download event is related to a (malware or benign) update, we examine the length of the download path and the time between requests. If the length of the total download path is < 4 or the time between requests is < 1 second, we consider the download event for detailed manual review. In this case, we analyze the HTTP transactions that precede the download by examining the content for artifacts, such as text and clickable buttons, that are indicators of human interaction. If none are found we label the download as an *update*.

Drive-by: Next we look for drive-by download indicators. To assist our labeling, we borrow some of the features proposed in [30]. Notice that the labeling of drive-by downloads is *not* a contribution of our paper. This is only a means to an end. The novel contributions of this paper are related to studying the characteristics of SE download attacks.

To label drive-by attacks, we look for “exploitable content,” such as pdf, flash, or java code on the path to a malware download. Browser plugins and extensions that process this type of content often expose vulnerabilities that are exploited by attackers. If we suspect the download event under analysis is a drive-by, we reverse engineer the content of the HTTP transactions that precede the suspected attack. This typically requires deobfuscating javascript and analyzing potentially malicious javascript code. For instance, if we identify code that checks for vulnerable versions of browser software and plugins (an indication of “drive-by kits”), we label the download as *drive-by*. We identify and label 26 drive-by downloads.

Social Engineering: If the cluster is not labeled as *update* or *drive-by*, we further examine the download events to determine if they are due to SE attacks. For this analysis, we inspect the content of all HTTP transactions on the download path. This content was saved at the time of the download and does not require revisiting of the URLs on the download path. Because SE downloads are attacks on the user, they are initiated by a user-browser interaction (e.g., clicking a link). Therefore, our goal is to identify the page on the download path containing the link likely clicked by the user that ultimately initiated the executable file download. By manually reviewing the web content included in the download path, we attempt to determine if *deception* or *questionable persuasion* tactics were used to trick the user into downloading the executable file (see Section 3). In case of positive identification of such tactics, we label the cluster as *social engineering*; otherwise, we label it as “likely” benign.

Notice that the analysis and labeling of SE download attacks is mainly based on the identification of deception tactics to trick a user to download an executable file. However, we also use AV scanning for confirmation pur-

poses. By doing so, we found that the majority of the clusters we label as social engineering contained one or more of downloaded files that were labeled as malicious by some AVs. This provides additional confirmation of our labeling of SE download attacks.

Overall, among 1,205 clusters in our dataset, we labeled 136 clusters as *social engineering*. In aggregate, these clusters included a total of 2,004 SE download attacks. In Section 3 we analyzed these SE download attacks and developed a categorization system that allows us to organize these attacks based on the deception and persuasion tactics used to attack the user. In the next section, we measure the popularity of these tactics based on the data we collected.

5 Measuring SE Download Attacks

In this section we measure the popularity of the tactics attackers employ to gain the user’s attention and of the deception and persuasion techniques that convince users to (unknowingly) download malicious and unwanted software. In addition, we measure properties of ad-based SE download attacks, which can be used to inform the development of effective defenses against SE attacks that leverage ad campaigns.

5.1 Popularity of SE Download Attacks

Table 1 shows the number and percentage of SE download attacks for each tactic employed by the attackers to gain the user’s attention. Over 80% of the SE attacks we observed used *ads* displayed on websites visited by the user. An additional 7% employed both *search* and *ad*, whereby the user first queries a search engine and is then presented with targeted ads based on the search terms. The popularity of ads in SE download attacks is likely due to the fact that they are a very efficient way for attackers to reach a large audience, thus maximizing the number of potential victims. Furthermore, well-crafted targeted ads are naturally highly effective at attracting a user’s attention.

Table 1: Popularity of SE techniques for gaining attention.

User’s Attention	Total	Percentage
Ad	1,616	80.64%
Search+Ad	146	7.29%
Search	127	6.34%
Web Post	115	5.74%

Gaining the user’s attention is not sufficient for an SE download attack to succeed. A user must also be tricked into actually “following the lead” and downloading the malicious or unwanted software. Table 2 shows the popularity of the deception and persuasion techniques we observed in our dataset of SE download attacks. The most

popular combination of deception and persuasion techniques is *repackage+entice*, making up over 48% of the observations. In most of these cases, the user is offered some type of “free” software of interest (e.g., a game or utility). Unfortunately, while the free software itself may not be malicious, it is often bundled with malicious applications such as *adware* or PUPs.

Table 2: Popularity of SE techniques for tricking the user.

Trick	Total	Percentage
Repackage+Entice	972	48.50%
Invent+Impersonate+Alarm	434	21.66%
Invent+Impersonate+Comply	384	19.16%
Repackage+Decoy	155	7.74%
Impersonate+Decoy	46	2.30%
Impersonate+Entice+Decoy	12	0.60%
Invent+Comply	4	0.20%
Impersonate+Alarm	1	0.05%

The next two most popular combinations of deception and persuasion tactics are *invent+impersonate+alarm* and *invent+impersonate+comply*, comprising 22% and 19% of the SE downloads we observed. An example of *invent+impersonate+alarm* is a Fake Java update attack, whereby the user is shown an ad that states “WARNING!!! Your Java Version is Outdated and has Security Risks, Please Update Now!” and uses images (e.g., logos or icons) related to Java (notice that this and all other examples we use throughout the paper represent real-world cases of successful SE attacks from our dataset). Ads like this are typically presented to users while they are visiting legitimate websites. In this example, the attacker is *inventing* the scenario that the user’s Java VM is out-of-date, *alarming* them with “WARNING!!!” displayed in a pop-up ad, and then *impersonating* a Java update that must be installed to resolve the issue. Notice that this is different from *repackage+entice*, in that the real Java software update is never delivered (only the malware is). Furthermore, the attacker leverages alarming messages about a well-known software to more aggressively “push” the user to download and install malicious software.

The difference between *invent+impersonate+alarm* and *invent+impersonate+comply* is in the persuasion component; i.e., *alarm* vs. *comply*. *Alarm* leverages fear (e.g., the computer may be compromised) to compel the user to download and install malicious software. On the other hand, *comply* leverages a pretend requirement necessary to complete a desired user task. For instance, a user may be presented with an ad on a video streaming website that says “Please Install Flash Player Pro To Continue. Top Video Sites Require The Latest Adobe Flash Player Update.” In this example, the attacker is *inventing* the need to install “Flash Player Pro” and tells the user they must *comply* before they can continue with watching the desired video. Unfortunately, this results

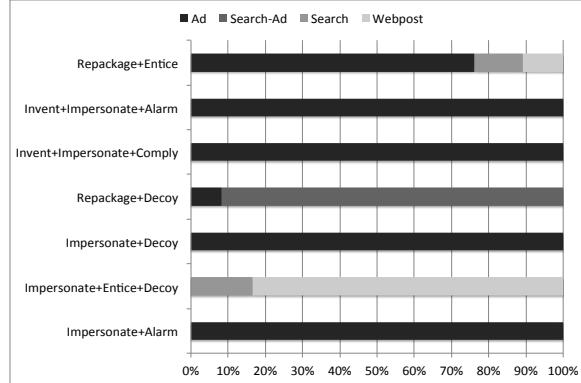


Figure 3: How attackers gain the user’s attention per deception/persuasion technique.

in the user downloading malicious software that simply *impersonates* a popular benign application and offers no actual utility to the user.

Table 3: Popularity of different “scam” tactics in the Ad:Invent+Impersonate subclasses.

	Alarm	Comply
Fake Flash	68%	20%
Fake Java	30%	0%
Fake AV	1%	0%
Fake Browser	1%	0%
Fake Player	0%	80%

Table 3 shows the popularity of different “scam” tactics in the invent+impersonate subclasses *alarm* and *comply*. Fake Flash and Java updates are the two most popular in the *alarm* class. In this same class we also observe Fake Browser updates and Fake AV alerts, but they are much less common, each comprising only about 1% of our observations. Fake Flash updates are also common in the *comply* class; however, the most popular scam tactic is telling the user they must update or install a new video player before they can continue. In these Fake Player attacks, images that resemble Adobe Flash Player are often used, but the terms “Adobe” or “Flash” are not directly mentioned. Therefore in Table 3 we distinguish between explicit Fake Flash and Fake Player.

Figure 3 shows how attackers gain the user’s attention for each of the observed deception and persuasion techniques. For instance, ads are the most common way used to attract users’ attention for repackage+entice, comprising 75% of our observations. *Search* and *web posts* contribute the remaining 25%. All observations for invent+impersonate+alarm, invent+impersonate+comply, impersonate+alarm and impersonate+decoy rely exclusively on ads to gain the user’s attention. At the other extreme, none of our observations for impersonate+entice+decoy use ads. This is likely due to the fact that this combination of deception and persuasion

techniques is more effective when the user’s attention is gained through *search* and *web posts*. However, notice that this comprises less than 1% of all SE downloads in our dataset (see Table 2).

5.2 Ad-based SE Download Delivery Paths

As shown in Table 1, the majority of SE attacks we observed use online ads to attract users’ attention. To better understand these attacks, we examine the characteristics of their *ad delivery path*. We begin by reconstructing the web path (i.e., the sequence of URLs) followed by the victims to arrive to the download URL (see Section 4). Then, we identify the first ad-related node on the web path using a set of regular expressions derived from the Adblock Plus rules [1]. We define the set of nodes (i.e., HTTP transactions) on the web path beginning at the first ad node and ending at the download node as the *ad delivery path*.

Table 4: Top five ad entry point domains.

Comply	Alarm	Entice
26% onclickads.net	16% adcash.com	20% doubleclick.net
10% adcash.com	7% onclickads.net	16% google.com
10% popads.net	7% msn.com	12% googleadservices.com
7% putlocker.is	6% yesadsrv.com	11% msn.com
3% allmyvideos.net	4% yu0123456.com	8% coupons.com

Table 4 shows the top 5 “entry point” domain names on the ad delivery paths (i.e., the first domain on the ad paths) for the *comply*, *alarm* and *entice* attack classes. Almost 50% of the ad entry points for the *comply* class begin with one of the following domains: `onclickads.net`, `adcash.com` or `popads.net`. By investigating these domains, we found that they have also been abused by adware in the past. Specifically, these domains are the source of pop-up ads injected into the user’s browsing experience by several well known adware programs and ad-injecting extensions [43].

Table 4 also shows that the top two ad entry points for the *alarm* class are the same as the *comply* class, though in reverse ranking. The third domain is `msn.com`, which has a good reputation. However, this domain is appearing at the top of the ranking probably because it sometimes redirects (via syndication) to less reputable ad networks, which in turn direct the user to an SE download. Notice also that the top entry domains in the *entice* class all have very good reputations (`doubleclick.net` is owned by Google). This is likely due to the fact that the majority of downloads in this class are for legitimate software that is simply bundled with “less aggressive” PUPs.

Besides performing an analysis of the “entry point” to the ad delivery path, we also analyze the last node on the path, namely the HTTP transaction that delivers the download. Table 5 shows the most popular SE download domains for the *comply*, *alarm* and *entice* classes. We

Table 5: Top five ad-driven SE download domains.

Comply	Alarm	Entice
17% softwaare.net	7% downloaddabs.com	41% imgfarm.com
5% newthplugin.com	4% downloaddado.com	17% coupons.com
5% greatsoftfree.com	4% whensoftisupdated.net	11% shopathome.com
4% soft-dld.com	3% safesystemupgrade.org	5% crusharcade.com
3% younplugin.com	3% onlinelivevideo.org	3% ilivid.com

found that the domains listed for the *comply* and *entice* classes serve mostly adware and PUPs.

To better understand the network-level properties of SE downloads, we also measure the “age” of these domains by leveraging a large passive DNS database (pDNS-DB) that stores historic domain name resolutions. Specifically, we define the domain age as the difference in days from the time the domain was first queried (i.e., first recorded in the pDNS-DB) to the day of the download. All the domains in Table 5 that are part of the *comply* and *alarm* classes are less than 200 days old, with the majority being less than 90 days. On the other hand, the domains in Table 5 for the *entice* class are all at least several years old. This is because most of the downloads in this class are for legitimate software that is simply bundled with adware or PUPs. For instance, we find a large variety of “free” software ads that direct users to the domain `imgfarm.com` for download. This is the reason that over 40% of the downloads in the *entice* class are from that domain.

The “middle of path” domains, namely the ones between the ad entry point and the download domain itself, tend to be a mix of recent and old domains. In fact, the most popular *comply* and *alarm* class “middle of path” domains are a 50/50 split of recent and old. However, this is not the case for the *entice* class, for which most domains are several years old. At the same time, the majority of ad delivery paths for all three classes include at least one middle domain name with an age that is less than 200 days.

5.3 Ad-Driven Benign Downloads

As mentioned earlier, more than 80% of the SE download attacks we observed use ads to gain the user’s attention (see Table 1). Based on common experience, it may seem unlikely that many benign software downloads would result from clicking on an ad. As a result, one may think that if software is downloaded after clicking on an ad, that software is unlikely to be benign. Somewhat surprisingly, we found that this may not necessarily be the case, as we explain below.

First, to automatically identify ad-driven benign software downloads, we first derive a set of ad detection regular expressions from the rules used by the popular AdBlock Plus browser extension [1]. We match these regular expressions against the nodes on the reconstructed download path for each benign download (the down-

load labeling process is described in Section 4.4). If an AdBlock rule matches the download path, we label that benign software download as *ad-driven*. We find that around 7% of all benign software downloads are ad-driven. Even though the percentage is low compared to SE downloads, in absolute terms this number is significant because the vast majority of software downloads observed in a network are benign. In fact, by considering the overall number of confirmed malware and benign software downloads and how many of these downloads are ad-driven, we find that if a software download is reached by clicking on an ad there is a 40% chance that that software is benign.

Table 6: Ad-based benign download popularity.

Category	Percentage
Games	32%
Utilities	30%
Music	15%
Business	11%
Video	8%
Graphics	2%
Social	2%

To further understand what type of benign software downloads are ad-driven, we investigate through manual analysis 100 randomly selected samples of benign software download events. Table 6 shows the type of software that is represented in our sample and their relative popularity. Games and utilities are the most popular categories, comprising 62% of all downloads. For example, the game “Trion Worlds” is the most popular with 17 downloads, followed by “Spotify” with 10 downloads.

6 Detecting SE Download Attacks

In this section, we measure the antivirus (AV) detection rate for SE downloads and group them into broad malware classes using AV labels. Also, we show that it is possible to accurately detect SE download attacks by constructing a statistical classifier that uses features derived from the SE attack measurements we presented in Section 5.

6.1 Antivirus Detection

To assess how AV products cope with SE downloads, we scan each SE download sample in our dataset with more than 40 AV engines (using VirusTotal.com). We scanned each of the samples on the day we collected them. Then, we also “aged” the samples for a period of one month, and rescanned them with the same set of AV engines. If at least one of the top five AV vendors (w.r.t. market share) and a minimum of two others detect it, we label the executable as malicious.

We first group malicious downloads into three broad classes, namely *malware*, *adware* and *PUP*. These

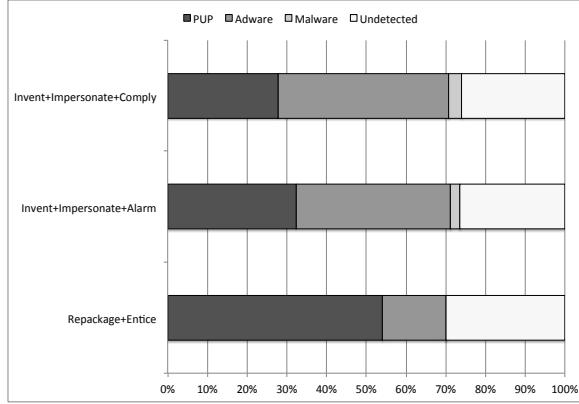


Figure 4: AV detections one month after download.

classes are meant to roughly indicate the potential “aggressiveness” of the malicious software. We assign these class labels based on a conservative set of heuristics that consist of matching keywords on the labels provided by the AVs. For instance, to identify adware we look for the string “adware” as well as the names of several popular adware applications (e.g., “amonetize,” etc.). Similarly, for the PUP class we look for the strings such as “PUP”, “PUA” and popular PUP application names. If both PUP and adware match, we label the sample based on a majority voting rule (across the different AV vendor labels). In the case of a tie, we conservatively label the sample as PUP. The remaining samples are labeled as malware, and manually reviewed for verification purposes.

Figure 4 shows the percentage of attacks that result in the download of malware, adware, and PUP, respectively. We show a breakdown for the top three deception/persuasion categories, which in aggregate represent more than 95% of all ad-driven SE attacks. The majority of malicious downloads in the invent+impersonate+comply and invent+impersonate+alarm deception/persuasion tactics belong to the *adware* class. A smaller percentage of downloads in these categories, 3.2% and 2.4% respectively, are labeled as *malware*. For repackage+entice the majority of downloads are labeled as *PUP*; no malware is found in this deception/persuasion category.

Notice that the relatively small percentage of *malware* is likely due to the fact that our heuristics for grouping malicious downloads into broad classes is very conservative, because it requires only a single AV label to contain an “adware” or “PUP” string to label the sample as belonging to the adware or PUP classes (see Section 4.4). In addition, adware is simply much more prevalent than malware, making the malware set size look relatively small compared to the rest of the dataset. Lastly, notice that only 70% – 75% of all malicious executables were labeled by the AVs, even after “aging” and rescanning the

samples. The remaining 25% – 30% is therefore labeled as *undetected* in Figure 4.

Table 7 shows the AV detection rate on the day of download for malware, adware and PUPs. Namely, we consider all SE download samples that were detected by AVs after aging and rescanning (i.e., after one month), and show the percentage of these samples that were also detected on the day we first observed them being downloaded. As can be seen, the detection rate in this case is quite small, thus confirming the reactive nature of AV detection.

Table 7: Zero day AV detections.

	Malware	Adware	PUP
Invent+Impersonate+Comply	0%	0%	6.9%
Invent+Impersonate+Alarm	5.9%	11.6%	26.9%
Repackage+Entice	NA	28.7%	34.4%

6.2 SE Detection Classifier

Guided by our measurements around SE attacks that we presented in Section 5, we devise a set of statistical features that can be used to accurately detect ad-driven SE downloads. We focus on ad-driven attacks because they are responsible for more than 80% of all SE downloads we observed (see Table 1).

Problem Definition. Given an executable file download event observed on the network, we first automatically reconstruct its *download path*, i.e., the sequence of pages/URLs the user visited to arrive to the HTTP transaction carrying the file, as explained in Section 4. Then, given a set of labeled ad-driven SE download events (Section 4.4) and also ad-driven benign software downloads (Section 5.3), we first translate the *download path* of each event into a vector of statistical features. Finally, we use the obtained labeled dataset of feature vectors to train a statistical classifier using the Random Forest [10] algorithm that can detect future ad-driven SE download attacks and distinguish them from benign ad-driven software downloads.

Statistical Features. We now present the set of detection features we derived and the intuitions behind their utility. In the following, we assume to be given as input the download path related to a software download event observed on the network, which we translate into a feature vector. Notice that no single feature by itself enables accurate detection; it’s their combination that allows us to reach high accuracy.

- **Ad-Driven** (binary feature). We check whether the download path contains an ad-related URL. This feature is computed by matching AdBlock [1] rules against the sequence of URLs on the download path.
Intuition: while the majority of SE downloads are

promoted via advertisement (Section 5), only 7% of benign downloads result from clicking on an ad (Section 5.3).

- **Minimum Ad Domain Age.** We measure the age of each domain on the *ad path*, namely the subsequence of the download path consisting of ad-related domains, and use the minimum age across these domains. *Intuition:* ad-serving domains that consistently direct users to malicious ads are often blacklisted, so they move to new domains. In essence, this feature is a way of (approximately) measuring the reputation of the *ad path*. Our measurements show that the majority of ad paths for the *comply*, *alarm* and *entice* attack classes all have domains less than one year in age. For benign download paths, this is true in only less than 5% of the cases.
- **Maximum Ad Domain Popularity.** Using our dataset (Section 4), we first consider all ad-related domains involved in the download paths observed in the past (i.e., in the training set). Then, for each domain, we count the number of distinct download paths on which the domain appeared, for both ad-driven SE attacks and the benign download paths. If the domain is found in more than 1% of the benign download paths, it is discarded. Otherwise, we compute the number of distinct SE attack paths in which the domain appeared. Finally, given all ad-related domains in the download path we are currently considering, we take maximum number of times a domain along this path appeared in an SE attack path. *Intuition:* some ad networks, and the domains from which they serve ads, are more abused than others, e.g., due to scarce policing of ad-related fraud and abuse in lower-tier ad networks. Therefore, they tend to appear more frequently in the download path of SE downloads. For instance, Table 4 in Section 5.2 shows the popularity of *ad entry points* for SE downloads.
- **Download Domain Age.** We measure the number of days between the download event and the first time we observed a DNS query to the effective second level domain for the download URL (final node of the web path) using a large historic passive DNS database. *Intuition:* the vast majority of benign downloads are delivered from domains that have been active for a long time because it takes time for a website to establish itself and attract visitors. On the other hand, SE domains are often “young” as they change frequently to avoid blacklisting. Our data shows that the download domain of over 80% of the invent+impersonate SE subcategories *comply*

and *alarm* are less than one year in age, whereas for benign download this only holds in 5% of the cases.

- **Download Domain Alexa Rank.** We measure the Alexa rank of the domain that served the software download. We compute this features using the effective second level domain for the download URL and the Alexa top 1 million list. *Intuition:* malicious executables are more likely to be hosted on unpopular domains because of their need of avoiding blacklisting. Conversely, benign software downloads are often hosted on popular domains. For instance, measurements on our data show that over 60% of the benign downloads are from domains with an Alexa rank in the top 100,000. On the other hand, the more “aggressive” SE downloads, such as those from the *alarm* class, are primarily delivered from very unpopular domains (very few are in the top 1 million). At the same time, the domains involved in SE attacks that trigger the download of PUP fall somewhere between, in terms of domain popularity.

6.3 Evaluating the SE Detection Classifier

In this section, we present the results of the evaluation of our SE detection classifier. We start by describing the composition of the training and test dataset, and then present an analysis of the false and true positives.

Datasets. To measure the effectiveness of the SE classifier, we use two separate datasets. The first dataset, \mathcal{D}_1 , which we use to train the classifier, consists of the software downloads described and measured in Sections 4 and 5. Specifically, this dataset includes 1,556 SE download paths (we consider all ad-driven SE attacks from the dataset described in Section 4), and 11,655 benign download paths.

The second dataset, \mathcal{D}_2 , consists of new executable downloads (and their reconstructed download paths) that we collected from the same deployment network in the *three months following* the completion of the measurements we presented in Section 5. Notice also that, \mathcal{D}_2 was collected *after* the feature engineering phase and after building our detection classifier. Namely, both the feature engineering and the training of the classifier were completed with *no access* to the data in \mathcal{D}_2 . Overall, \mathcal{D}_2 contains 1,338 ad-driven SE downloads, and 9,760 benign download paths. We label \mathcal{D}_2 following the steps outlined in Section 4.4.

Classification Results. After training our SE detection classifier using dataset \mathcal{D}_1 and the Random Forest learning algorithm, we test the classifier on dataset \mathcal{D}_2 .

Table 8 reports the confusion matrix for the classification results. The classifier correctly identified over 91% of the ad-driven SE downloads. Furthermore, it has a very low false positive rate of 0.5%.

Table 8: Confusion matrix for the SE detection classifier.

	Predicted Class	
	Benign	Ad-Based SE
Benign	99.5%	0.5%
Ad-Based SE	8.8%	91.2%

Table 9: SE Subclass Performance.

	True Positives
Repackage+Entice	65%
Invent+Impersonate+Alarm	98%
Invent+Impersonate+Comply	90%

Figure 9 shows a breakdown of the classification results for the subclasses of ad-based SE downloads. The invent+impersonate+alarm and invent+impersonate+comply categories have 98% and 90% true positive rates, respectfully. The lower performance for repackage+entice is due to downloads of legitimate software bundled with PUPs from well established domains. Because these domains are “mixed use,” and have high popularity or Alexa ranking, they make the detection task more difficult.

Feature Importance. We estimate feature importance by performing forward feature selection [20]. The single feature that provides the largest information gain is *download domain age*. Using only that feature we have a 69% true positive rate and a 6% false positive rate. By adding *maximum ad domain popularity*, we obtain a true positive rate above 80% with less than 3% false positives. As we add other features (using the forward search), both the true positives and false positives continue to improve. Thus, all the features help achieve high accuracy.

7 Discussion

In this paper, we focus exclusively on *successful* web-based SE download attacks (we consider the attacks we collected and study successful because they actually trigger the delivery of malicious software to the victim’s machine). Social engineering attacks carried over different channels (e.g., email) and that have different objectives (e.g., phishing attacks to steal personal information, rather than malware infections) are not part of our measurements, and are therefore also not reflected in the categories of SE tactics we described in Section 3. However, we believe ours is an important contribution. In fact, as defenses for drive-by downloads continue to improve (e.g., through the hardening of browser software and operating system defenses) we expect the attackers to increasingly make use of web-based SE attacks for malware propagation. Therefore, the reconstruction and analysis of SE download attacks is important because in-the-wild SE attack samples could be used to better train users and mitigate the impact of future attacks; thus, complementing automatic attack detection solutions.

Our study relies on visibility over HTTP traffic and

deep packet inspection. One might think that the inability to analyze HTTPS traffic represents a significant limitation. However, it is important to take into account the following considerations. When a user browses from an HTTPS to an HTTP site, they are often redirected through an unsecured intermediate URL, so that the `Referer` field can be populated with the domain and other information related to the origin site [3]. Alternatively, the origin site can set its referrer policies [2] to achieve the same result without need of intermediate redirections. As an example, even though many of the searches performed using search engines such as Google, Yahoo and Bing occurred over HTTPS, we were able to identify the search engines as the origin of web paths because the related domain names appeared in the `Referer` field of the subsequent HTTP transactions. Furthermore, modern enterprise networks commonly employ SSL man-in-the-middle (MITM) proxies that decrypt traffic for inspection. Therefore, our SE attack detection system could be deployed alongside SSL MITM proxies.

Throughout the study, we use the term *malicious* to describe the software downloaded as the result of an SE attack. However, there exist many shades of maliciousness and some malicious software (e.g., ransomware, botnets, etc.) are more “aggressive” than others (e.g., adware and PUPs). Therefore, in several parts of the analysis we broke down our results by distinguishing between *malware*, *adware* and *PUPs*. As shown in Section 6.1, only a relatively small percentage of the SE downloads collected for our measurements were categorized by our AV-label-based heuristics as *malware*. The majority were labeled as adware or PUP. However, we should notice that AV labels are known to be noisy and that our labeling heuristics are very conservative (see Section 4.4). Furthermore, over 25% of the malicious downloads remained unlabeled due to lack of AV detection (Section 6.1). Therefore, it is possible that the number of malware is somewhat higher than reflected in Section 6.1. However, the categorization system, network-level properties and detection results for SE attacks that deliver adware apply to attacks that result in malware downloads as well.

While the software downloads and traffic we collected for our study were collected from a single academic network, we should consider that the deployment network was very large, serving tens of thousands diverse users, consisting of users from different ages, cultures and backgrounds.

Because our SE detection classifier is designed to detect ad-based SE download attacks, an attacker could evade the system by using tactics other than online ads to attract the user’s attention (e.g., *search* or *web post*, as discussed in Section 3). However, advertisements are the

predominant tactic used by attackers because they allow them to “publish” their SE attacks on sites that already popular with the targeted victims. In addition, ads are only shown to the users that “match” their delivery criteria, thus reducing exposure to others (including security researchers) that could result in the discovery and mitigation of these attack vectors.

Another way an attacker may try to evade detection, is to specifically attempt to evade our statistical features (see Section 6.2). For instance, to evade the *download domain age* and *domain Alexa rank* features, the attacker could host the malicious files on a free file sharing site. This could result in a download domain with an age > 1 year and a high Alexa ranking. However, the *ad-driven*, *minimum ad domain age* and *maximum ad domain popularity* features, which are harder for the attacker to control, could still allow to identify most attacks. For example, simply knowing that a software download resulted from an online ad puts its probability of being malicious at more than 50%, according to the real-world data we collected (see Section 5.3). Furthermore, if hosting malicious downloads on free hosting sites became popular, then a *Free File Hosting* feature could be added to our feature set, as it is unlikely that many ad-driven benign software downloads are served from free file hosting sites.

8 Related Work

Social engineering is primarily an attack on users, not systems. The fundamental concepts that are employed to exploit the user are rooted in modern psychology, specifically in the study of persuasion [13] and deception [41]. SE attacks have been studied in [21, 27]. While these works study SE tactics in general, they do not focus on SE download attacks. To the best of our knowledge, the only systematic study on SE malware is [8], which discusses the psychological and technical ploys adopted by SE attacks and some trends in SE malware. However, [8] focuses on malware that spreads via e-mail and on SE tactics used by malware to lure the user to activate (i.e., run) the malicious code. In addition, the data analyzed in [8] is limited to malware attack case studies and statistics published by others in the *VirusBulletin* journal until 2010. In contrast, we focus on web-based SE downloads, and on reconstructing and analyzing how users are tricked into downloading malicious software in the first place. Also, our analysis is based on recent real-world instances of successful SE attacks collected from a live network.

Malware downloads have been studied in a number of works [11, 22, 34, 39]. For instance, [34, 39] use a content-agnostic detection approach that relies on computing a reputation score for the domains/IPs from which malware downloads are served. However, [34, 39] are

generic malware download detection systems that offer no understanding of what caused a malware download in the first place. In other words, they cannot identify the origin of the attack, but only its side effects (i.e., only the malware download even itself), and therefore offer no clue on whether an infection was caused by a SE attack. Other works focus on the properties of malware droppers [11, 22], whereby already infected machines download malware updates or new malware strains. In contrast, we study how users fall victim to web-based SE download attacks, and design a detection system that can accurately detect ad-driven SE downloads.

Researchers have also separately examined specific types of SE malware attacks, such FakeAVs [15, 16, 19, 25, 37]. Our work is different because we propose a general approach to studying, measuring and classifying SE download attacks on the web. We do not limit ourselves to specific attack types such as Fake AVs. Therefore, our work has broader applications, and also provides mitigation against generic ad-based SE download attacks, which represent a large percentage of all SE download attacks we observed in the wild.

Other works have focused on traffic redirection chains to understand and detect malicious websites and attack delivery [23, 26, 38]. Among these systems, Mad-Tracer [23] studies malicious advertisement, including ad chains that deliver malware downloads. This is done by crawling popular websites, and using a supervised classifier trained on data labeled by leveraging domain name blacklists (e.g., Google SafeBrowsing). While part of our work, namely our ad-driven SE download detection system, also leverages some properties of advertisement chains to detect ad-driven malicious downloads, it is important to notice that we focus specifically on *in-the-wild SE download attacks* and are able to identify a large variety of SE download attacks. For instance, [23] only reports fake anti-viruses (AVs) as malware delivered via ad-based *scams*. Instead, in our study we find many other types of SE-driven downloads that leverage a variety of deception and persuasion tactics. In fact, our measurements show that fake AVs represent only a small fraction (less than 1%) of all SE attacks. In addition, rather than actively looking (or crawling) for possible malware downloads on popular websites, we collect *live SE attacks* by directly witnessing successful attacks against users in a large academic network. This allows us to collect *successful SE attacks*, rather than *possible attacks* as done in [23].

In our work, we use a combination of web traffic reconstruction and analysis to trace back the origin of the attacks, namely the SE tactic that tricked the user into downloading malicious software. Researchers have studied web traffic reconstruction in [12, 29, 30, 42]. Among these, the closest to our work is WebWitness [30], a re-

cently proposed incident investigation system that aims to provide context to malicious downloads by reconstructing the path taken by the user to download executable files. WebWitness is able to classify the cause of a malicious download as *drive-by*, *social engineering* or *update*. In [30], the main focus is on studying drive-by downloads and developing a new defense against drive-by download attacks. However, social engineering attacks are not studied. WebWitness is able to separate drive-by downloads from social engineering downloads once an *oracle* identifies a download as malicious, and is not able to independently detect SE attack. Although we utilize WebWitness' trace-back algorithm, our contributions are very different from [30], because we study SE download attacks in depth, focusing on their collection, analysis and categorization, as well as the detection and mitigation of ad-based SE-driven infections.

9 Conclusion

In this paper, we presented the first systematic study of social engineering (SE) attacks that trigger software downloads. To this end, we collected and reconstructed more than two thousand examples of in-the-wild SE download attacks captured at a large academic network. We performed a detailed analysis and measurements on the collected data, and developed a categorization system to identify and organize the tactics typically employed by attackers to make SE download attacks successful. Furthermore, by measuring the characteristics of the network infrastructure used to deliver such SE attacks, we were able to engineer a number of features that can be leveraged to distinguish between SE and benign (or non-SE) software downloads with a true positive rate of 91% at a false positive rate of only 0.5%.

References

- [1] AdBlock Plus. <https://adblockplus.org/>.
- [2] ReferrerPolicies. <https://www.w3.org/TR/referrer-policy/#referrer-policy-origin>.
- [3] Protecting privacy with referrers, 2010. <https://www.facebook.com/notes/facebook-engineering/protecting-privacy-with-referrers/392382738919/>.
- [4] The download.com debacle: What CNET needs to do to make it right, 2011. <https://www.eff.org/deeplinks/2011/12/downloadcom-debacle-what-cnet-needs-do-make-it-right>.
- [5] Huge decline in fake av following credit card processing shakeup, 2011. <http://krebsonsecurity.com/2011/08/huge-decline-in-fake-av-following-credit-card-processing-shakeup/>.
- [6] Fake virus alert malware (fakeav) information and what to do, 2013. <http://helpdesk.princeton.edu/kb/display.plx?ID=1080>.
- [7] Heres what happens when you install the top 10 download.com apps, 2015. <http://www.howtogeek.com/198622/heres-what-happens-when-you-install-the-top-10-download.com-apps/?PageSpeed=noscript>.
- [8] ABRAHAM, S., AND CHENGALUR-SMITH, I. An overview of social engineering malware: Trends, tactics, and implications. *Technology in Society* 32, 3 (2010), 183 – 196.
- [9] BOTT, E. Social engineering in action: how web ads can lead to malware, 2011.
- [10] BREIMAN, L. Random forests. *Mach. Learn.* 45, 1 (Oct. 2001).
- [11] CABALLERO, J., GRIER, C., KREIBICH, C., AND PAXSON, V. Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC’11, USENIX Association, pp. 13–13.
- [12] CHEN, K. Z., GU, G., ZHUGE, J., NAZARIO, J., AND HAN, X. Webpatrol: Automated collection and replay of web-based malware scenarios. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS ’11, ACM, pp. 186–195.
- [13] CIAUDINI, R. B. *Influence: Science and Practice*, 5th ed. Pearson Education, 2000.
- [14] COVA, M., KRUEGEL, C., AND VIGNA, G. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web* (New York, NY, USA, 2010), WWW ’10, ACM, pp. 281–290.
- [15] DIETRICH, C. J., ROSSOW, C., AND POHLMANN, N. Exploiting visual appearance to cluster and detect rogue software. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2013), SAC ’13, ACM, pp. 1776–1783.
- [16] DUMAN, S., ONARLIOGLU, K., ULUSOY, A. O., ROBERTSON, W., AND KIRDA, E. Trueclick: Automatically distinguishing trick banners from genuine download links. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New York, NY, USA, 2014), ACSAC ’14, ACM, pp. 456–465.
- [17] GRIER, C., BALLARD, L., CABALLERO, J., CHACHRA, N., DIETRICH, C. J., LEVCHENKO, K., MAVROMMATHIS, P., MCCOY, D., NAPPA, A., PITSLIDIS, A., PROVOS, N., RAFIQUE, M. Z., RAJAB, M. A., ROSSOW, C., THOMAS, K., PAXSON, V., SAVAGE, S., AND VOELKER, G. M. Manufacturing compromise: The emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS ’12, ACM, pp. 821–832.
- [18] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP ’08, IEEE Computer Society, pp. 402–416.
- [19] KIM, D. W., YAN, P., AND ZHANG, J. Detecting fake antivirus software distribution webpages. *Comput. Secur.* 49, C (Mar. 2015), 95–106.
- [20] KOHAVI, R., AND JOHN, G. H. Wrappers for feature subset selection. *Artificial Intelligence* 97, 12 (1997), 273 – 324. Relevance.
- [21] KROMBHLZ, K., HOBEL, H., HUBER, M., AND WEIPPL, E. Advanced social engineering attacks. *J. Inf. Secur. Appl.* 22, C (June 2015), 113–122.
- [22] KWON, B. J., MONDAL, J., JANG, J., BILGE, L., AND DUMITRAS, T. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS ’15, ACM, pp. 1118–1129.

- [23] LI, Z., ZHANG, K., XIE, Y., YU, F., AND WANG, X. Knowing your enemy: Understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 674–686.
- [24] LU, L., YEGNESWARAN, V., PORRAS, P., AND LEE, W. Blade: An attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 440–450.
- [25] MAVROMMATHIS, P., BALLARD, L., PROVOS, N., INC, G., AND ZHAO, X. The nocebo effect on the web: An analysis of fake anti-virus distribution. In *In USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)* (2010).
- [26] MEKKY, H., TORRES, R., ZHANG, Z.-L., SAHA, S., AND NUCCI, A. Detecting malicious http redirections using trees of user browsing activity. In *INFOCOM, 2014 Proceedings IEEE* (April 2014), pp. 1159–1167.
- [27] MITNICK, K. D., AND SIMON, W. L. *The Art of Deception: Controlling the Human Element of Security*, 1st ed. John Wiley & Sons, Inc., New York, NY, USA, 2002.
- [28] NAPPA, A., RAFIQUE, M. Z., AND CABALLERO, J. Driving in the cloud: An analysis of drive-by download operations and abuse reporting. In *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2013), DIMVA'13, Springer-Verlag, pp. 1–20.
- [29] NEASBITT, C., PERDISCI, R., LI, K., AND NELMS, T. Click-miner: Towards forensic reconstruction of user-browser interactions from network traces. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 1244–1255.
- [30] NELMS, T., PERDISCI, R., ANTONAKAKIS, M., AND AHAMAD, M. Webwitness: Investigating, categorizing, and mitigating malware download paths. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2015), SEC'15, USENIX Association, pp. 1025–1040.
- [31] POWER, R., AND FORTE, D. Social engineering: attacks have evolved, but countermeasures have not. *Computer Fraud and Security* 2006, 10 (2006), 17 – 20.
- [32] PROJECT, T. C. Out-of-process iframes (OOPIFs). <https://www.chromium.org/developers/design-documents/oop-iframe>.
- [33] PROVOS, N., MCNAMEE, D., MAVROMMATHIS, P., WANG, K., AND MODADUGU, N. The ghost in the browser analysis of web-based malware. In *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets* (Berkeley, CA, USA, 2007), HotBots'07, USENIX Association, pp. 4–4.
- [34] RAJAB, M. A., BALLARD, L., LUTZ, N., MAVROMMATHIS, P., AND PROVOS, N. Camp: Content-agnostic malware protection.
- [35] RAJAB, M. A., BALLARD, L., MAVROMMATHIS, P., PROVOS, N., AND ZHAO, X. The nocebo effect on the web: An analysis of fake anti-virus distribution. In *Proceedings of the 3rd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More* (Berkeley, CA, USA, 2010), LEET'10, USENIX Association, pp. 3–3.
- [36] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, ACM, pp. 219–232.
- [37] STONE-GROSS, B., ABMAN, R., KEMMERER, R. A., KRUEGEL, C., STEIGERWALD, D. G., AND VIGNA, G. The underground economy of fake antivirus software. In *In Proc. (online) WEIS* 2011 (2011).
- [38] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 133–144.
- [39] VADREVU, P., RAHBARINIA, B., PERDISCI, R., LI, K., AND ANTONAKAKIS, M. Measuring and detecting malware downloads in live network traffic. In *Computer Security ESORICS 2013*, J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 556–573.
- [40] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 417–432.
- [41] WHALEY, B. Toward a general theory of deception, 1982. Military Deception and Strategic Surprise.
- [42] XIE, G., ILIOFOTOU, M., KARAGIANNIS, T., FALOUTSOS, M., AND JIN, Y. Resurf: Reconstructing web-surfing activity from network traffic. In *IFIP Networking Conference, 2013* (2013), IEEE, pp. 1–9.
- [43] XING, X., MENG, W., LEE, B., WEINSBERG, U., SHETH, A., PERDISCI, R., AND LEE, W. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th International Conference on World Wide Web* (New York, NY, USA, 2015), WWW '15, ACM, pp. 1286–1295.

Specification Mining for Intrusion Detection in Networked Control Systems

Marco Caselli
University of Twente
m.caselli@utwente.nl

Emmanuele Zambon
University of Twente & SecurityMatters B.V.
emmanuele.zambon@secmatters.com

Johanna Amann
ICSI
johanna@icir.org

Robin Sommer
ICSI & LBNL
robin@icir.org

Frank Kargl
Ulm University
frank.kargl@uni-ulm.de

Abstract

This paper discusses a novel approach to specification-based intrusion detection in the field of networked control systems. Our approach reduces the substantial human effort required to deploy a specification-based intrusion detection system by automating the development of its specification rules. We observe that networked control systems often include comprehensive documentation used by operators to manage their infrastructures. Our approach leverages the same documentation to automatically derive the specification rules and continuously monitor network traffic. In this paper, we implement this approach for BACnet-based building automation systems and test its effectiveness against two real infrastructures deployed at the University of Twente and the Lawrence Berkeley National Laboratory (LBNL). Our implementation successfully identifies process control mistakes and potentially dangerous misconfigurations. This confirms the need for an improved monitoring of networked control system infrastructures.

1 Introduction

Starting from Denning’s seminal work in 1986 [9], *intrusion detection* has evolved into a number of different approaches. Among them, *anomaly-based intrusion detection* and, most recently, *specification-based intrusion detection* have gained attention for their potential to detect previously unknown attacks (e.g., zero-day attacks).

A specification-based intrusion detection system (IDS) leverages functional specifications of a system to model its properties, or *features*, creating a reference of correct behavior. Differently from anomaly-based IDSs, behavior of features is not derived by a learning phase (prone to false positives) but directly extracted from documentation. This ensures the quality of the generated models and improves detection. Several research efforts in the literature confirm the accuracy of

specification-based intrusion detection [62]. However, these approaches assume manual (human) analyses, often focused just on network protocol documentations.

Scaling a specification-based approach to an entire infrastructure faces three key challenges. First, targeted systems do not always have consistent features, and thus *constraints*, that allow to describe functional rules (e.g., common networks do not usually guarantee stable communication patterns or message timing). Second, even when such constraints are present, bridging the semantic gap between infrastructure properties and the low-level features actually observable by an IDS remains hard [22]. Third, deploying a specification-based intrusion detection requires an explicit and unambiguous description of the features’ behaviors, as well as substantial human effort in crafting the related specification rules.

This work aims to fill this last challenge by automating specification-based intrusion detection to a fairly high degree. We propose an approach to automatically mine IDSs’ specification rules from available documentation. Our approach works under the following assumptions:

- Documentation about monitored systems must be *available*. No specification-based intrusion detection would be possible without relying on correct information that describes an infrastructure’s components, mechanisms and constraints. Documentation should be provided in an electronic form to allow automated knowledge extraction.
- Information retrieved from the documentation must be *linkable* to what an IDS can observe. On a network, information about components, mechanisms and constraints need to map to features monitored by the IDS (e.g., information on a whitelist of network services should link to the correct IP address).

We do not claim that these principles can be generically applied to any system. However, we observe that environments such as Networked Control Systems

(NCSs) [20] suit this approach. NCSs are “systems whose constituents such as sensors, actuators, and controllers are distributed over a network, and their corresponding control-loops are formed through a network layer” [30]. Examples of NCSs include: industrial control systems [65], building automation systems [42] and in-vehicle networks [31]. NCSs generally have the properties discussed above. Communications over these networks are quite stable [22] (e.g., neither the number of devices nor the way data is shared change regularly). Moreover, automation guarantees the presence of control algorithms and consequently the existence of consistent features that will eventually become the core of the specification rules. These features and their constraints also represent the most attractive target for an attacker who wants to manipulate the controlled processes [14].

The two assumptions defined above hold for NCSs. Information related to system features are often documented in configuration files, reference books and manuals (e.g., “Substation Configuration Language” files for industrial control systems, “Protocol Implementation Conformance Statement” for building automation systems, CAN matrixes and corresponding documentation for in-vehicle networks). Linkability is generally guaranteed by the absence of encryption and by the verbosity of the adopted protocols.

In this paper, we design and discuss a specification-based network intrusion detection system (NIDS) for BACnet-based building automation systems to demonstrate and investigate our concept. First, we present specification-based intrusion detection in §2. Then, we introduce building automation and BACnet in §3. We outline our approach in §4. Details of the approach in the context of building automation systems are discussed in §5 to §7. Finally, we examine the general applicability of our work in §8.

2 State of the Art

Ko et al. introduce specification-based intrusion detection in [34]. The authors describe their approach towards automated detection of Unix privileged program misuses and suggest to “specify programs’ intended behavior” by modeling their normal execution beforehand. The proposed solution works through the definition of a Program Policy Specification Language aiming to formally define programs’ operations by simple predicate logic and regular expressions. Later works such as [13] resume and improve the proposed ideas. Ko et al. improve their introductory research in [35] by defining a formal framework used to define and detail security-relevant behavior of Unix programs. In [33], the framework gets integrated into a comprehensive specification-based IDS, called SHIM. SHIM merges several different detection

approaches that apply to both network communications and operating system activities. Its use, together with machine learning techniques, was shown to be an effective solution towards the development of automated intrusion response strategies [3]. From the previous works, Sekar et al. continue developing the research field by proposing complementary approaches in [52] (based on the use of the Auditing Specification Language) and [54] (based on a custom language called “Regular Expressions for Events” or REE). The same authors, present in [53] a hybrid approach aiming to increase the information of a specification-based IDS with the use of anomaly-based intrusion detection techniques. The authors use Extended Finite State Automata (EFSA) to model and detail network protocol behavior. Then they refine the set of monitored features via a learning phase exploiting statistical analyses on the traffic traces.

Over the years, researchers customized specification-based intrusion detection to fit different infrastructures such as mobile ad hoc networks [60, 44, 23, 56, 61, 19] and WLANs [16]. Furthermore, specification-based IDSs were developed for specific use cases both for network-based (e.g., VoIP technologies [59], carrier Ethernet [26]) and host-based security (e.g., kernel dynamic data structures [48], mobile operating systems [7]).

Specification-based intrusion detection has gained a main role in NCSs. Works such as [8], [27], and [37] present specification-based IDSs for Modbus, Zigbee and DNP3 respectively. Hadeli et al. notably apply a semi-automated specification approach to substation automation systems employing MMS and GOOSE [21]. The authors leverage operator input to parse infrastructure-related documentation and derive security checks. Ultimately, Berthier et al. take this approach a step forward by modeling not just employed protocols (in this case C12.22) but smart-meter security constraints and policies as well [4]. This research shows the feasibility and effectiveness of modeling high-level infrastructure properties.

State-of-the-art research on specification-based intrusion detection assumes that protocol and system documentation is readily available when designing and configuring the IDS. Few thoughts are spent on where and how to retrieve this information, especially not in an automated way. Moreover, all the aforementioned works do not explore *the possibility of autonomously extracting the information needed to build the related IDSs from documentation*, instead relying on human evaluation and translation to rules. This possibility would allow to efficiently apply specification-based approaches to entire infrastructures. As discussed in the introduction, our research focuses on this key aspect of specification-based intrusion detection with the aim of making its development more time-effective and accurate.

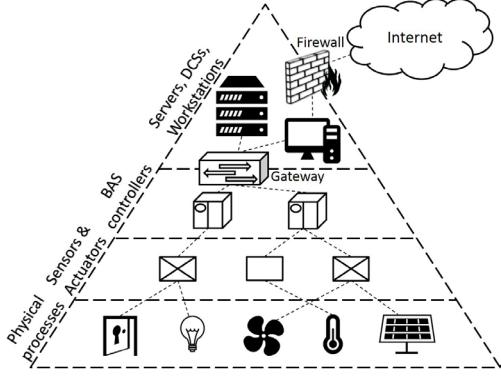


Figure 1: Building automation network layout

3 Case Study: Building Automation

Building automation systems (BASs) or building management systems (BMSs) are networked infrastructures controlling operations and services within a building (or a group of buildings). Among other uses, building automation systems can monitor and control HVAC (heating, ventilation, and air conditioning), lighting, energy consumption, and physical security and safety [42].

A building automation network usually follows a hierarchical layout [28] (Figure 1). At the bottom, sensors and actuators directly connect to the monitored physical processes and send information back and forth to building automation controllers. Controllers communicate with servers and distributed control systems (DCSs) to coordinate high-level control procedures and policies. Finally, operators can access and manage building automation components connecting through their workstations and human-machine interfaces (HMIs).

In the last decade, the employment of building automation solutions has constantly increased (both for commercial and residential buildings) and its market share is expected to grow in the following years [39]. Despite numerous benefits (e.g., energy efficiency, “smart homes”, etc.) building automation makes several new threat scenarios not just feasible but realistic [24, 17, 18, 47]. Nevertheless, only few solutions have been proposed to improve building automation system security [18, 6, 45].

3.1 BACnet

The “Building Automation and Control Network” (BACnet) protocol [1] facilitates building automation system communication for a wide array of different devices and different settings. While exact statistics of the proliferation of BACnet are difficult to come by, already back in 2003 there were more than 28,000 BACnet installations in 82 countries [28].

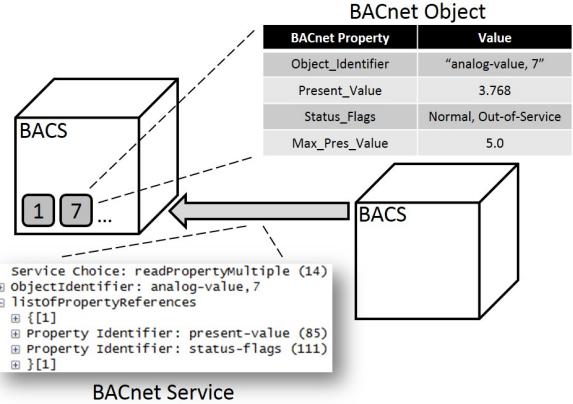


Figure 2: BACnet interaction example

BACnet has a layered protocol architecture, similar to the ISO/OSI model. The BACnet protocol has an application layer, containing the actual application data payload as well as a network layer that abstracts the differences of the network architectures supported by BACnet and implements its own routing protocol. Underneath the network layer, BACnet also specifies how it can be used with different types of data links.

The BACnet application layer rests on two important core concepts: *objects* and *services* (Figure 2). A Building Automation and Control System (BACS)¹ includes one or more BACnet objects that are used to represent its functions. Objects are of a specific type, like Analog Input or Analog Output. BACnet supports a wide range of high-level object types like Calendars, Date Value objects, or Credential Data Input objects. BACnet users and vendors can define “proprietary” objects as well to serve specific functionalities. Object types have different attributes that are called properties, which are extensible for specific purposes. The second core concept of the BACnet application layer are services. While objects describe the different functions that are implemented by a BACS, services define how to communicate with the BACS, offering functionality such as reading object information from a device. Their names reflect the semantics of the operation (e.g., ReadProperty).

Individual BACSS typically only support a small selection of possible objects and services. Manufacturers use a “Protocol Implementation Conformance Statement” (PICS) to describe which objects and services are implemented by a specific device. The BACnet standard implies that all BACSS shall have a PICS identifying “all of the portions of BACnet that are implemented” [1]. Information in PICS includes: a brief description of the de-

¹In the remainder of this paper, we will refer to any controller implementing BACnet as BACS.

vice; a list of supported BACnet Interoperability Building Blocks (BIBBs) that define classes of BACnet services supported by a device; and a list of supported standard and proprietary BACnet objects and properties with their characteristics.

To complete device descriptions, operators may take advantage of configuration files such as “Engineering Data Exchange” (EDE) files [2]. Generally, operators compile these documents to represent internal characteristics of a deployed BACS. EDE files include detail information on devices’ BACnet implementations (e.g., which BACnet objects a device is currently using) and value constraints (e.g., in Figure 2, Present_Value of Analog Value 7 must be less than 5.0).

3.2 Attacks on BACnet

BACnet defines a limited security architecture providing peer and operator authentication along with data confidentiality and integrity (“Clause 24 — Network Security” [1]). However, none of this is implemented in available products [43]. This leaves BACnet infrastructures vulnerable to numerous cyber-threats [24, 63, 58, 29].

We categorize attacks on BACnet into three main groups: snooping, denial of service (DoS), and process control subverting. This categorization derives from the list of BACnet protocol threats described in [24].

Snooping attacks concern stealing information about a specific building automation system. To achieve this goal, these attacks require access to the building automation system network. Once inside, attackers can take advantage of BACnet services such as `ReadProperty` and `ReadPropertyMultiple` to gain knowledge of the BACS. This includes device models, locations, status, and information on their BACnet support (e.g., which BACnet services and objects they implement). Attackers may need this information to understand the infrastructure and pave the way to further intrusions. However, snooping attacks do not disrupt any process of the building automation system.

Differently, DoS attacks try to interfere with control processes by making controllers unreachable for operators. This category only considers DoS attacks that are performed through the use of BACnet routing features (e.g., malicious modifications to the BACnet routing tables) and leaves other kinds of DoS out of its scope. As for the snooping attacks, DoS attacks need malicious users to have access to the network. Moreover, this kind of attacks requires information about the network layout. Attackers can achieve their goal by sending BACnet messages, such as `Initialize-Routing-Table`, to modify a BACS’ routing tables. In this way, operators lose visibility on single devices or even entire sections of the building automation system.

Finally, process control subverting includes those attacks that directly modify control processes and, consequently, interfere with physical operation. This kind of attacks requires more skilled attackers with sufficient knowledge about the building automation system functioning. In this scenario, attackers exploit specific controllers by using several different BACnet services, such as `WriteProperty` or `DeleteObject`, to change the BACSs’ structures and operations. This leads to a loss of control by the operators and, consequently, leads to risks for components and people.

3.3 Evaluation Environments

For this work, we analyzed two different building automation installations over more than two months of constant operation. The first building automation system belongs to the University of Twente in the Netherlands and is in charge of supervising utilities and services provided to the university campus. Its duties encompass energy consumption control, HVAC, and room monitoring and management (e.g., pressure and temperature control, shading, etc.). The second building automation system belongs to the Lawrence Berkeley National Laboratory (LBNL) and supervises several services on its premises. The LBNL process control focuses mostly on room monitoring and energy consumption for the Lab facilities. Both infrastructures deal with hundreds of BACSs from several different vendors.

The IDS we deployed at the University of Twente linked to a SPAN port on a switch directly connected with the SCADA servers monitoring the whole building automation system. The same switch is responsible for routing most of the traffic of the building automation network. This allowed us to capture and analyze most of the BACnet messages exchanged by BACSs. Differently, at LBNL we could monitor only a subset of the building automation system by linking to a switch in charge of connecting BACSs inside one building. However, this was sufficient to automatically gather the information needed for our approach to craft the specification rules.

The two infrastructures generally showed similar traffic patterns. Several BACSs shared the same sets of objects and used the same kind of messages to exchange information. Furthermore, both UT’s and LBNL’s traffic samples included numerous BACnet routing messages (e.g., `Who-Is`, `I-Am`, `Who-Has`, and `I-Have`) organizing communication paths within the two networks. However, the two infrastructures presented some differences related to communication and control strategies (e.g., all BACSs deployed at UT used confirmed services thus requiring acknowledgments from message recipients while some devices at LBNL used just unconfirmed ones). Particularly, the employment of

BACSSs from different vendors led operators to employ individual procedures implemented through the use of `ConfirmedPrivateTransfer` BACnet services. Such services are used to invoke proprietary or non-standard routines in remote BACSSs.

3.4 Setting and Threat Model

The reason to develop a network-based IDS is twofold. First, a network-based solution is easier to deploy than host-based ones. Secondly, this setup allows us to have minimum impact on NCS processes. Once deployed, we assume that our system is able to capture real-time traffic of the monitored building automation system in a completely passive fashion and to retrieve documentation publicly available on the Internet. This allows to gather the information we need to build specification rules and implement effective detection.

On the other side, we assume attackers can gain full access to the network as well. We consider this happening in a way that is similar to standard IT environments (e.g., phishing, software vulnerability exploitation). Tools such as Shodan [40] show how easy it is to find building automation networks exposing their devices to the Internet. Once inside, attackers can obtain a convenient viewpoint on the building automation control processes. Two key factors support this assumption. First, most building automation protocols take advantage of broadcast communications to exchange information among devices (e.g., routing notifications). This already allows attackers to easily observe a large part of the traffic. Secondly, the hierarchical structure of common building automation networks steers valuable information messages towards servers and DCSs. By gaining access to one of these servers, attackers can observe most of the traffic within the building automation system.

Within a building automation network, attackers may use attacks outlined in §3.2 to gain knowledge on, or subvert, the correct functioning of the building automation system. In this last scenario, any safety feedback in place can usually be overridden [49]. Therefore, attackers can put infrastructure components under stress, possibly threatening human safety when it comes to devices such as electrical equipment.

4 Specification Mining Approach

Our approach works towards automated development of specification rules for network security monitoring. Setting up and customizing a specification-based IDS for a particular infrastructure requires a large amount of information about the monitored system, implying a substantial manual effort in gathering and refining the specification rules. As details of the infrastructures are often

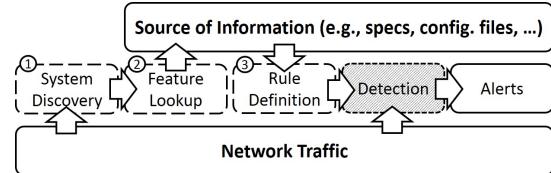


Figure 3: Specification-mining approach

described within specs and configuration files, especially in many NCS environments, the process of collecting this information—and, consequently, the development of the actual IDS—can be automated to a fairly high degree through the following steps (see also Figure 3):

(1) System Discovery gathers information about the monitored NCS. In this step, our system analyzes the network traffic in order to: 1) identify devices communicating on the network (e.g., models, brands); and 2) determine role and purpose of each identified device (e.g., a device is a controller, an HMI, etc.). Every time the system collects enough information about a specific device it proceeds with the next step.

(2) Feature Lookup implements a set of information retrieval techniques to gather knowledge about devices identified during System Discovery. The purpose of this step is to: 1) find verified information (e.g., specs, configuration files) about the infrastructure’s devices; and 2) select features and constraints from the retrieved documents and arrange results in a structured form.

A successful Feature Lookup relies on the assumptions of *availability* and *linkability* outlined in the introduction. The assumption of *availability* implies the existence of documents about infrastructures and components that are automatically retrievable. This requires the information to be provided in electronic form and being suitable for parsing. Also, this assumption includes an assurance on the authenticity of the retrieved information (e.g., by the use of reliable sources, by the employment of secure retrieval techniques). The assumption of *linkability* guarantees that the information derived by the retrieved documentation can be checked by the system against observations within the traffic (e.g., messages, variables, etc.). Particularly, after the identification of network devices and the successful retrieval of their related constraints from the documentation, the assumption of *linkability* enables assigning effective specification rules to the right targets.

(3) Rule Definition uses the knowledge obtained in the Feature Lookup to craft the specification rules. To achieve this goal, the system needs to: 1) select identified information from Feature Lookup; 2) translate this information to specification rules.

We focus our specification-based intrusion detection on *controllers* (e.g., BACSSs for building automation systems, Programmable Logic Controllers or PLCs for ICSs, Electronic Control Units or ECUs for in-vehicular networks). This decision comes from the key role these components have within NCSs: Controllers are involved in any monitoring and control operation of the infrastructure either autonomously or accessed by operators. Furthermore, controllers are likely targets for attackers (as illustrated in §3.2).

We observe that NCS controllers share a number of properties. First, every controller *employs a limited set of variables to fulfill its function*. These variables can go from simple memory addresses to complex objects but *often have predetermined types*. Moreover, all controllers *use a limited set of methods (or services) to access and manipulate variables of other controllers*. Finally, *each variable can assume a limited range of values according to its type or the physical characteristic it represents*. We leverage these shared properties to define a set of general constraints, or *abstract rules*, checking NCS variables' *types*, *values* and access *methods*. These abstract rules are the seeds we use to automatically generate specialized specification rules. To achieve this, we define a mechanism that maps information retrieved in the Feature Lookup step to the abstract rules. This process automatically completes the abstract rules and, as a result, customizes detection for the monitored NCS.

Once a rule is defined, it becomes active and, thus, part of the detection mechanism. During detection, an active rule verifies if its related constraint is fulfilled or not. When this last condition becomes true, the system triggers an alert for the user.

Having presented the phases in a generic way, we now describe our experimental setup and, then, how we have instantiated them to build a specification-based IDS for BACnet-based building automation systems.

Implementation background We implement our approach using the Python programming language [51] and Bro [46]. Bro is a network traffic analyzer employed in different domains such as network security monitoring and performance measurement. The system comes with comprehensive built-in functionality for traffic analysis and supports several network protocols ranging from standard (e.g., HTTP, FTP) to domain-specific (e.g., Modbus [41], DNP3 [10]). Bro provides a Turing-complete scripting language that allows users to select and analyze network events (e.g., connection establishments). We choose to describe specification rules through the “Bro scripting language” because of its efficiency and expressiveness. We developed a BACnet parser for Bro using Spicy [55], a parser generator whose specification language allows users to define a protocol’s

syntax and semantics at a high level. We publish the BACnet parsing code for Spicy, as well as the Python scripts , as open source software.² However, we cannot open-source the Bro code containing the rule checks due to privacy agreements with the two building automation system sites.

5 System Discovery

To identify BACSSs we implement three different techniques that we term: “*BACnet Device Object analysis*”, “*BACnet Address linking*”, “*BACnet Property set fingerprinting*”. The first technique directly follows from the protocol standard and relies on the mandatory presence of a *Device* object in every BACS device. The *Device* object defines “a standardized object whose properties represent the externally visible characteristics of a BACnet device”. Among these properties there are: *Object_Name*, *Vendor_Name*, *Vendor_Identifier*, *Model_Name*, *Firmware_Revision*, *Application_Software_Version*, *Location*, and *Description*. Most of these properties are set by vendors and provide information on a device’s identity (e.g., *Model_Name*) and role (e.g., *Description*). BACnet services such as *ReadProperty* and *ReadPropertyMultiple* can access those properties. As these services are widely employed by user interfaces and logging servers to automatically update data related to infrastructure’s components, information on *Device* objects regularly passes through the network and, thus, is available to System Discovery. As the *Object_Identifier* property of a *Device* object is a parameter that uniquely identifies a device in a BACnet network, a message such as the one in Figure 4 allows us to identify a BACS and understand its purpose. In the Wireshark screenshot example, BACS with identifier “17001” is a “Blue ID S10 Controller”.

For BACnet objects of other types, since no information can be extracted from the IP address (multiple BACnet devices may share the same IP address), a further parameter allows to identify message sources and destinations: the BACnet address. As for the *Device* object’s *Object_Identifier*, the BACnet address (together with the *Network Identifier*) is unique within a BACnet network. In the “*BACnet Address linking*” technique, the BACnet address bridges the gap between a known *Device* object and any BACnet object included in the same BACS. Figure 5 shows an example of this analysis. When “device 4001” is known (as a result of the previous technique), any message carrying both the related *Device* object’s *Object_Identifier* and the

²<https://github.com/specification-mining-paper-usenix-2016/specification-mining>

```

Service Choice: readPropertyMultiple (14)
ObjectIdentifier: device, 17001
listOfResults
  [1]
  ...
  Property Identifier: vendor-identifier (120)
  {[4]}
  vendor-identifier: (unsigned) 105
  {[4]}
  ...
  Property Identifier: model-name (70)
  {[4]}
  model-name: UTF-8 'Blue ID S10 Controller'
  {[4]}
  ...

```

Figure 4: “BACnet Device Object analysis” example

```

Building Automation and Control Network NPDU
Version: 0x01 (ASHRAE 135-1995)
Control: 0x08
Source Network Address: 4000
Source MAC Layer Address Length: 6
SADR: a1:0f:00:00:00:00 (a1:0f:00:00:00:00)
Building Automation and Control Network APDU
0001 .... = APDU Type: Unconfirmed-REQ (1)
Unconfirmed Service Choice: i-Am (0)
ObjectIdentifier: device, 4001
...
```

(a) I-Am message

```

Building Automation and Control Network NPDU
Version: 0x01 (ASHRAE 135-1995)
Control: 0x08
Source Network Address: 4000
Source MAC Layer Address Length: 6
SADR: a1:0f:00:00:00:00 (a1:0f:00:00:00:00)
Building Automation and Control Network APDU
0011 .... = APDU Type: Complex-ACK (3)
.... 0000 = PDU Flags: 0x00
Invoke ID: 216
Service Choice: readProperty (12)
ObjectIdentifier: analog-value, 171
...
```

(b) ReadProperty message

Figure 5: “BACnet Address linking” example

BACnet Address allows us to link the two parameters (Figure 5a). Any later message then carrying the BACnet address along with a further object (e.g., “Analog Value 171”) enables linking to the corresponding device (Figure 5b). This technique works well because I-Am messages pass the network frequently to ensure visibility of all BACnet objects.³

Finally, if no information can be extracted from Device objects or BACnet addresses, System Discovery can benefit from observations of the BACnet properties. As discussed in §3.1, the BACnet property set is

³The technique can also directly use messages carrying Device object information if the source BACnet address is present in the header. However, for this kind of messages, having the BACnet Address fields is not mandatory.

Table 1: University of Twente - BACS device list

# of devices	Vendor	Model	Role
5	Kieback&Peter	DDC4000	DCS
15	Priva	HX 80E	Router
7	Priva	Compri HX	Controller
25	Priva	Compri HX 3	Controller
36	Priva	Compri HX 4	Controller
12	Priva	Compri HX 6E	Controller
85	Priva	Compri HX 8E	Controller
2	Priva	Blue ID S10	Controller
16	Priva	Comforte CX	HMI
2	Delta Controls	eBCON	Controller
3	Siemens	PXG80-N	Controller
3	Siemens	PXC64-U	Controller
3	Siemens	PXC128-U	Controller
3	Siemens	PXR11	Controller
3	Siemens	PXC00-U + PXA30-RS	Controller
1	Unknown	Unknown	-

Table 2: LBNL - BACS device list

# of devices	Vendor	Model	Role
23	Automated Logic	LGR	Router/Gateway
14	Automated Logic	ME	Controller
11	Automated Logic	SE	Controller
159	Automated Logic	ZN	Controller
1	Automated Logic	WebCTRL	HMI
9	Johnson	NAE	Controller
1	Johnson	NIE	Controller
4	Paragon Controls Inc.	EQ	Controller
4	Sierra	BTU Meter	Energy meter
4	Sierra	FFP	Controller
1	Tracer	UC400	Controller
2	Niagara	AX Station	SCADA server
7	Unknown	Unknown	-

extensible. Every object of a BACS has a set of standard and proprietary properties that form a “fingerprint” of that object and device. The third technique assumes that two objects sharing the same fingerprint are likely to be of the same kind. During System Discovery, it is possible to create a database of identified fingerprints each one pointing to the corresponding BACS (identified with the previous two techniques). Whenever an unknown object presents a property set already in the database, the system infers the most likely related device.

Experiments Previous work by us shows that traditional fingerprinting techniques are usually ineffective on most NCSs [5]. In our tests, tools such as Nmap [38] and PoF [64] were able to identify just a limited number of Windows and Linux workstations. The techniques presented above proved more effective. Thanks to frequent ReadPropertyMultiple, our system was able to gather information on most BACSs. Moreover, BACnet address linking and BACnet property set fingerprinting allowed the system to link most of the observed BACnet objects to identified devices. At the end of System Discovery, we gathered information on ~15k BACnet objects belonging to the 445 devices shown in Tables 1 and 2.

Thanks to the information from the operators, we

know that we correctly identified 98.2% of the BACSSs actually deployed (445 out of 453 devices). Eight devices did not link to any useful BACnet message or identifiable property set. However, these devices convey almost no information over the network (a few hundreds BACnet messages over two months of capturing compared to an average of tens of thousands) and did not involve any notable equipment. Identifying the aforementioned 445 devices took just a few hours of monitoring.

6 Feature Lookup

Searching for documentation on identified BACSSs is possible because the two assumptions of *availability* and *linkability* hold for BACnet-based building automation systems. Verified information about BACSSs is *available* within PICSs and EDE files. This information includes BACSSs' vendors, models and even refers to specific BACnet objects, thus is *linkable* to what we observed over System Discovery.

Feature Lookup targets both online and offline documentation. On the one hand, we use Google APIs to search and retrieve publicly available documents such as PICSs on the Internet. On the other hand, we retrieve EDE files from private repositories in the installations. Both cases allow for document authenticity. In the former case, we narrow the search to a subset of reliable sources such as vendors' websites and reputable third parties (e.g., BACnet International Laboratories⁴). In the latter case, we assume a secure connection to a trusted dataset managed by the operators.

Once a BACS links to one or more of these documents, our system parses the documents looking for useful information. According to BACnet specifications, a PICS has a standard template and we observe that most PICSs are closely modeled to it. Figure 6 shows three extracts from the PICS of the “Blue ID S10 Controller” mentioned in the previous section.

As outlined in §3.1, each PICS provides a description of the related BACS and the BIBBs it implements (Figure 6a). Moreover, PICSs include information about supported BACnet objects and properties, as well as their characteristics (Figures 6b and 6c).

EDE files also follow a standard template but they use a simpler “comma-separated values” (CSV) format. Each EDE file presents details of a specific BACS (Figure 7 shows an extract of Device 4001 EDE file). Data includes all implemented BACnet objects (e.g., “device 4001” owns “Analog Value 171”, “Multi-state Value 15”, etc.) and their descriptions. Furthermore, EDE files include information about Present_Value properties with

⁴<http://www.bacnetinternational.org/>

Product information	
Date	2013-07-23
Product name and model number	Blue ID S10 Controller
Application software version	1.0
Firmware revision	1.00
BACNet protocol revision	9
Product description	The Blue ID S10 Controller contains a powerful microprocessor. It can be easily programmed for building automation purposes. The processing speed and computing power mesh seamlessly with the requirements of modern and integrated systems. The controller uses a reliable operating system that ensures quality and operational security. It is fully controllable via BACnet including commissioning, writing, reading, alarm and event handling.
BACnet standardized device profile	BACnet Building Controller (B-BC)

Vendor information	
Vendor name	Priva
Vendor ID	105
Contact information	PO Box: 18 2578 ZG The Netherlands www.priva.co.uk

Supported BIBBs	
DS-RP-A	Data Sharing - Read Property - A
DS-RP-B	Data Sharing - Read Property - B
DS-RPM-A	Data Sharing - Read Property Multiple - A

(a) PICS excerpt 1

Standard object types supported ¹		Dynamically creatable and deletable
Accumulator	no	
Analog Input	no	
Analog Value	no	
Binary Input	no	
Binary Output	no	
Binary Value	no	
Calendar	yes	
Device	no	

(b) PICS excerpt 2

Supported properties per object type	Required or optional	Readable (R) or readable/writeable (R/W)	Additional comments
Accumulator			
Object_Identifier	required	R	
Object_Name	required	R	
Object_Type	required	R	
Present_Value	required	R/W	
Description	optional	R	
Creation_Status	required	R	

(c) PICS excerpt 3

Figure 6: PICS example

PROJECT_NAME	UTWENTEN4(Hoofdschema)						
VERSION_OF_REFERENCEFILE	126						
IMPLEMENTATION_DATE	14-04-2013 11:11:59 800						
AUTHOR_OF_LAST_CHANGE	Regio Partners						
VERSION_OF_LAYOUT	2						
device obj-instance	object-type	object-instance	present-value-default	min-present-value	max-present-value	unit-code	description
4001	AV - 171	3	171	0	0	100	98 Select KM1W1 - stuinsel
4001	MV - 15	19	15	1			Selectieve nieuwvSelect
4001	AV - 11	2	11	100	0	32767	71 Selectieve nieuwvBent
4001	AV - 42	21	42	0	0	2	95 Selectieve nieuwvBrug
4001	AV - 43	2	2	0	0	2	95 Selectieve nieuwvBrug

Figure 7: EDE file example

value ranges (e.g., “Analog Value 171” can vary from min-present-value 0 to max-present-value 100).

Experiments The program we implemented to search for online documentation uses the outputs of System Discovery (vendors and models) and further keywords such as “PICS” to retrieve information about identified BACSSs. The system ranks Google results coming from public repositories (e.g., www.bacnetinternational.net) and the web by quantifying the presence of the keywords in document titles. For example, the “Blue ID S10 Controller” links to a PDF document titled “BACnet_PICS_Blue_ID_S10_Controller.pdf” (Figure 6). With this technique we identified a PICS for 99.3% of the de-

vices deployed in the two building automation systems (442 out of 445 among the devices identified in the System Discovery step). Two ‘Siemens PXR11’ and one ‘Paragon Controls Inc. EQ’ were the only devices that did not link to any PICS. However, we could not find the related PICSs even by a manual search either.

Offline research targeted specific devices directly. While online documentation always provides general information about BACnests of a certain kind (e.g., all “Blue ID S10 Controller”), offline repositories provide detailed information related only to devices deployed in the monitored building automation system. For this reason, instead of vendors and models we searched through the available documents using device `Object_Instances`. For example, starting from “Device object 4001” from System Discovery, we found an EDE file titled “Controller 4001_EDE.csv”. While LBNL did not provide any configuration file, operators from the University of Twente shared with us 10 files of this kind. While they confirmed that there was indeed an EDE file for every deployed device, they could not grant us unlimited access to all of them due to information sensitivity. For this reason, the operators chose the 10 files based on roles and purposes of the related devices. Each file we obtained described a BACS identified over System Discovery.

The aforementioned privacy concerns refer to the initial manual analysis we had to perform over the EDE files and would not hamper the applicability of our approach. In an ideal deployment, one would have a secure connection between the IDS and the machine storing the EDE files, without any human activity involved for retrieval operations and processing. However, both University of Twente and LBNL operators store infrastructure documentation on computers also used for other purposes than building automation, and direct connections to those resources were infeasible.

Finally, we implemented two programs to parse PICSs and EDE files respectively. In the first case, the program goes from document’s top to bottom guided by the diagram shown in Figure 8. For every available PICS, the program first selects all implemented BIBBs and BACnet objects (Figures 6a and 6b). Each object can be creatable/deletable and this information follows the object as a “yes/no” or equivalent symbols (Figure 6b). Finally, for every object, the script selects a list of properties that can be writable or not (Figure 6c). Figure 9 shows parsing results of the “Blue ID S10 Controller” coming from the PICS showed in Figure 6.

Most of the retrieved PICS did not have any information about property values. Instead, this information was included in the EDE files. A further program went through all EDE files selecting `Present_Value` minimum and maximum values for every listed object. This new information was structured as shown in Figure 10.

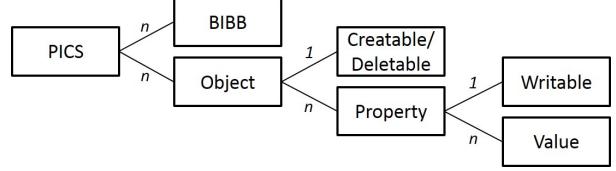


Figure 8: PICS parsing diagram

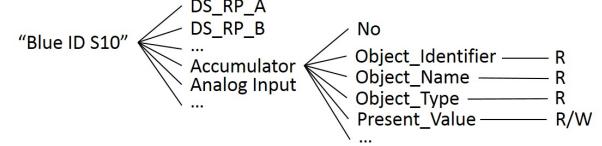


Figure 9: Parsing PICS example results

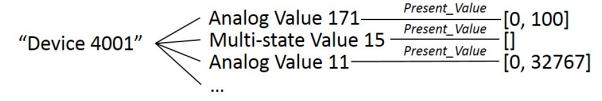


Figure 10: Parsing EDE example results

7 Rule Definition and Detection

Next we describe how the information gathered in previous steps is used to define specification rules. In §4, we motivated our focus on variables’ *types*, *values* and related access *methods* as basis for our specification rules. From this, we derive three *abstract* rules: 1) a “Type” rule checks if a variable of a specific type is allowed; 2) a “Value” rule checks which values a variable may assume; and 3) a “Method” rule checks which methods can be used to access a specific variable. All rules have the same structure: each element (type, value, method) is evaluated against a set of allowed possibilities. For example, in the “Type” rule, a variable’s type is evaluated against all the allowed types of variable a controller may implement (Algorithm 1).

We use a Python program to automate the process of mapping information retrieved over Feature Lookup to the abstract rules. In the following, we discuss how we map these abstract rules into specification rules for monitoring for each type.

Type Rule: The “Type” rule checks which BACnet objects and properties each BACS can use. This information comes from the PICSs (Figures 6b and 6c) and, thus, is included in the results of Feature Lookup (Figure 9). Therefore, a script selects allowed objects and properties of each identified BACS and transforms the “Type” rule into the two specification rules shown in Algorithm 2.

In the case of the “Blue ID S10 Controller”, the

Algorithm 1 Abstract “Type” rule

```
1: if Variabletype  $\notin$  ControllerAllowedVariableTypes then  
2:   Alert(“Variable type not permitted”)  
3: end if
```

Algorithm 2 BACnet “Type” rules

```
1: if BACnet Object  $\notin$  ControllerAllowedObjectTypes then  
2:   Alert(“Forbidden Object”)  
3: end if  
  
1: if BACnet Property  $\notin$  ControllerObjectAllowedPropertyTypes then  
2:   Alert(“Forbidden Property”)  
3: end if
```

Controller_{AllowedObjectTypes} set contains objects Accumulator, Analog Input, etc. In the same way, the Controller_{Accumulator}_{AllowedPropertyTypes} set of a “Blue ID S10 Controller” contains properties Object_Identifier, Object_Name, etc. Whenever the system captures a BACnet message including an object and some property, the two rules check object and property types respectively and alert if these types are not included in the defined sets. This allows the system to detect snooping attacks and any other attack dealing with unexpected objects and properties.

Value Rule: The “Value” rule checks which values BACnet properties may assume. This information comes from the EDE files (Figures 7 and 10) and, thus, is automatically mapped to the concrete rule as shown in Algorithm 3.

Algorithm 3 BACnet “Value” rule

```
1: if BACnet Property value  $\notin$  Controller(Object.Property)AllowedPropertyValues then  
2:   Alert(“Forbidden Value”)  
3: end if
```

For example, when it comes to “Device 4001”, the system alerts if “Analog Value 171” is below 0 or above 100. This rule protects the infrastructure against process control subverting scenarios, and thus attacks attempting to modify parameters of the physical and control processes.

Method Rule: The “Method” rule validates the BACnet services each BACS can use. This information comes from the PICSSs in the form of a list of BIBBs (Figures 6a) and is included in the results of Feature Lookup (Figure 9). BIBBs can be replaced with corresponding services by a simple lookup operation. Therefore, Algorithm 4 checks if a BACnet service belongs to the set of allowed services.

In the case of the “Blue ID S10 Controller”, the Controller_{AllowedServices} includes services from BIBBs DS_RP_A (ReadProperty_Request), DS_RP_B

Algorithm 4 BACnet “Method” rule

```
1: if BACnet Service  $\notin$  ControllerAllowedServices then  
2:   Alert(“Forbidden Service”)  
3: end if
```

(ReadProperty_Response), etc. This rule allows the system to detect attackers misusing BACnet services to fulfill their goals.

Furthermore, we use the “Method” rule to check which BACnet object is creatable/deletable and which BACnet property is writable. Following the standard, we compile three sets of BACnet services with services that create objects, that delete objects, and that write properties respectively. Then, the system uses the information from Feature Lookup to define checks on non-creatable/deletable objects and non-writable properties by using Algorithm 5.

Algorithm 5 BACnet additional “Method” rules

```
1: if BACnet Service  $\in$  CreateObjectServices then  
2:   Alert(“Forbidden object creation”)  
3: end if  
  
1: if BACnet Service  $\in$  DeleteObjectServices then  
2:   Alert(“Forbidden object deletion”)  
3: end if  
  
1: if BACnet Service  $\in$  WritePropertyServices then  
2:   Alert(“Forbidden property writing”)  
3: end if
```

For example, the first two rules alert the presence of services attempting to create or delete Accumulator objects belonging to a “Blue ID S10 Controller”. The third rule reports any service attempting to write to a non-writable property, such as Accumulator’s Object_Identifier.

Experiments Our system filled the abstract rules with the information coming from Feature Lookup crafting hundreds of specification rules. To improve efficiency we arrange the specification rules in an order that avoids meaningless checks (e.g., we do not want to check a BACnet property if we already know that the BACnet object it belongs to is not allowed). For every captured BACnet message, the system checks if the BACnet service is allowed; then, if involved BACnet objects can be used, created or deleted; then, if involved BACnet properties are allowed and writable. Finally, the system examines properties’ actual values. Only a small set of specification rules are of this last type due to the limited number of EDE files that operators provided us with.

As outlined in §3.3, we tested our approach against more than two months of real traffic. Over the two months of capturing, our system triggered 237 unique alerts; 226 at the University of Twente and 11 at LBNL.

Table 3: Detection results

Abstract Rule	Specification Rule	# Alerts
Type Rule	Forbidden object	2
	Forbidden property	234
Value Rule	Forbidden value	0
Method Rule	Forbidden service	0
	Forbidden object creation	0
	Forbidden object deletion	0
	Forbidden property writing	1

```
Service Choice: readPropertyMultiple (14)
④ ObjectIdentifier (214) Vendor Proprietary Value, 121
④ listOfPropertyReferences
④ {[1]}
④ Property Identifier: acked-transition (0)
④ ...
```

Figure 11: Unexpected object ReadProperty_Request

The two results differ because of the different views we achieved over the two infrastructures (as already described, at LBNL we could monitor only a subset of the building automation system and thus a subset of the traffic). Table 3 shows the three abstract rules, the corresponding specification rules and whether or not a rule raised an alert.

We did not find any evidence of malicious activities over the time span of the captures. However, our approach still provided interesting insights. At the University of Twente, the system raised alerts on two BACSs using forbidden objects. Both cases involved a “proprietary” object never described within the PICS. According to the available documentation, the two devices (two Siemens controllers PXC128-U) should not include anything that was not defined within the BACnet standard. Nevertheless, a device probed the two controllers (Figure 11) and received back correct BACnet responses about an unknown object. A meeting with the operators revealed that this object is vendor-defined and gathers information on parameters of the BACSs recognizable and understandable by vendors only. Operators confirmed that vendors have access to the building automation system to monitor their devices, and use of an unknown BACnet object happens even though the documentation does not mention this possibility because of its internal nature. However, operators did not know that involved BACSs provide specific functionalities and attackers can potentially exploit such circumstances.

Detection on BACnet properties provided the highest number of alerts (all alerts at LBNL were of this kind). Our system generated several alerts on ReadProperty and ReadPropertyMultiple messages attempting to retrieve non-existing properties. As a matter of fact, these properties were not defined by the PICSs and, for most cases, we could eventually confirm the

```
Service Choice: readPropertyMultiple (14)
④ ObjectIdentifier: device, 1050634
④ listOfPropertyReferences
④ {[1]}
④ Property Identifier: max-segments-accepted (167)
④ ...
```

(a) Unexpected property ReadProperty_Request

```
Service Choice: readPropertyMultiple (14)
④ ObjectIdentifier: device, 1050634
④ listOfResults
④ {[1]}
④ Property Identifier: max-segments-accepted (167)
④ propertyAccessError
④ {[1]}
④ error class: property
④ error code: unknown-property
④ ...
```

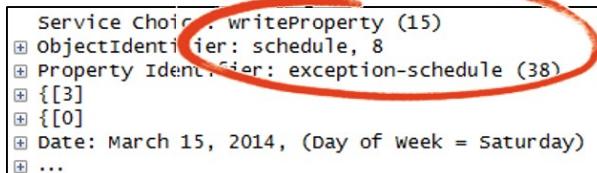
(b) ReadProperty_Response confirmation of the alert

Figure 12: Unexpected property read operation

non-existence of these properties by observing some BACnet errors carried in the responses to those read requests (Figure 12).

A BACS asking for unimplemented properties is not necessarily a violation of the specs. In fact, all PICSs define what a BACS implements without defining what other BACSs may ask for. A situation in which a BACS sends back a BACnet-error response to warn about a non-existing property (Figure 12b) is in line with the specs and should be of no harm for the system. However, the reason to alert on situations of this kind is twofold. First, this situation may be of interest from a security perspective. Despite being handled by the BACnet protocol, these circumstances may hide a “network discovery” scenario where an attacker tries to gain knowledge of the infrastructure by randomly probing BACSs. As described in §3.2, snooping is one plausible attack in building automation systems. Secondly, the same situation shows a common side-effect of the joint use of different BACnet software solutions. As servers and workstations do not usually know in advance which BACSs they will connect to, predefined BACnet discovery messages exist in order to gather general information of building automation components. These messages do not consider which BACnet properties are defined for each device and simply use large sets of them. This consequently generates several error responses on the network.

To dig deeper into property-related issues, we extended the “unknown property” specification rule to also check if properties enforced by PICSs were always implemented. Therefore, we created a further instance of “Type” rule checking all BACnet error messages to detect missing properties that were supposed to be used by the BACSs. The system revealed several messages reporting “unknown-property” errors about properties declared to be part of devices’ BACnet implementations.



```

Service Choice: writeProperty (15)
⊕ ObjectIdentifier: schedule, 8
⊕ Property Identifier: exception-schedule (38)
⊕ {[3]
⊕ {[0]
⊕ Date: March 15, 2014, (Day of week = Saturday)
⊕ ...

```

Figure 13: Unexpected property write request

All these mismatches between implementation and specification are particularly relevant for what concerns interoperability. In fact, software solutions that define their interactions with a BACS based on its public documentation can incur into inconsistencies caused by incorrect or lacking implementations.

Finally, the system triggered an alert corresponding to an unexpected write operation on a BACnet property supposed to be readable only. A Priva controller received a BACnet WriteProperty request on the `Exception_schedule` of an object `Schedule` (Figure 13). Despite what we knew from the related PICS, the BACS sent back a SimpleACK message, acknowledging the success of the operation (the actual writing was confirmed by later read operations). These kinds of situations are especially dangerous due to the unpredictability of their results. As no indication is provided by the vendor, the write operation can either succeed or fail, and may generate a response or not (even independently from the actual modification of the value within the property). Meeting with the operators revealed that this write operation was due to a human mistake during the configuration of the Priva controller. However, the same situation could fit the “process control subverting” scenario described in §3.2.

8 Discussion

Performed experiments confirm the feasibility of the approach within building automation systems and pave the way for its application to different domains.

8.1 Analysis of the Results

By construction, our IDS is able to detect events that do not match the specifications coming from retrieved documentation. This aspect leads to two considerations:

- On the one hand, an alert raised by the system does not necessarily refer to a security-relevant event as the related mismatches may not directly harm the monitored devices. However, all findings revealed network activities otherwise invisible to operators. Over the two months of analysis, every alert

identified either an actual mismatch between device documentation and implementation (e.g., unimplemented BACnet Properties) or an operator mistake (e.g., the unexpected writing operation). As already discussed, these issues can cause significant gaps in the knowledge operators have about their infrastructures and may potentially lead to dangerous misconfigurations of the involved systems. The meeting with the operators at the University of Twente confirmed that employed HMIs were not able to signal any of the misconfigurations found or even notify the users on generated BACnet errors. As a result, the University of Twente asked to deploy our system into the building automation system continuously and let operators receive notifications of the generated alerts. So while our datasets did not include actual attacks, we were able to reliably detect notable deviations from the specifications at zero false-positives. This result is in line with the work of Uppuluri et al. [62] showing that specification-based intrusion detection works towards optimal detection rate while substantially decreasing the number of false positives compared to anomaly-based detection.

- On the other hand, our approach does not necessarily detect all possible attacks threatening the monitored infrastructure. In fact, any attack operating within the boundaries defined by employed specifications would not be caught by our IDS. However, our solution substantially narrows down what a malicious user can do and covers most of the attack scenarios defined within the categories listed in §3.2. Furthermore, our solution does not exclude the use of other approaches such as pure anomaly-based intrusion detection either improving the obtained rule set or working in parallel.

Each one of the implemented phases effectively achieved the defined goals. Thanks to the numerous read operations, System Discovery took just a few hours of network sniffing to gather all the information needed to describe the whole set of BACSs. With this information, our approach was able to rapidly and automatically identify available sources of information and craft effective specification rules.

Feature Lookup focused just on structured documents such as PICSs and EDE files. In some of the tests, we further extended online research to documents such as BACS user manuals. Our system was able to download 10 manuals related to components deployed in the monitored infrastructures. However, we decided to not further employ manuals because an analysis showed they were fully overlapping with the information found in the PICSs. Nevertheless, one way to improve our

specification-mining approach is to enable handling heterogeneous documentation and, especially, unstructured information. To this regard, we observe that Feature Lookup should abstract from domain-specific parsing scripts and generalize the process of mining and structuring infrastructure features. Correctly selecting information can take advantage of standard data mining and natural language processing. Our work did not present a general approach to this activity. However, works such as [57, 15, 50] may fulfill this goal. With more general techniques capable of extracting knowledge from heterogeneous documentation, the effort of deploying the system completely converges on mapping retrieved information to the abstract rules. According to the monitored infrastructures, operators should identify the related concepts of variable type, value and access method and, eventually, let the system interpret data coming from Feature Lookup and instantiate the specification rules.

Even without such a general approach, our solution drastically reduced the time needed to deploy intrusion detection into a BACnet-based building automation system. Obtaining the same set of specification rules by hand would have required substantial effort, making it infeasible for larger infrastructures. Furthermore, the obtained system comes with the intrinsic capability to update according to the changes of the monitored infrastructure. In fact, whenever new BACSs are deployed, our system transparently reads the new information over the network and goes through the three steps all over again. In the end, this solution makes the implemented system directly applicable to any other BACnet infrastructure with no further effort on configuration or deployment.

The proposed approach works likewise for different building automation technologies. As discussed, this would mostly require a modification of the mapping process linking retrieved information and abstract rules but would leave the core concept unchanged. Other building automation infrastructures such as KNX [32] and LonWorks [11] also meet the requirements of *availability* and *linkability*. These widely used protocols present characteristics similar to the ones observed for BACnet. Moreover, both KNX and LonWorks promote and support the use of documents describing protocol implementation details (although not as formal as BACnet PICS).

To show the generality of our approach beyond building automation systems we outline how the same specification-mining technique applies to two different domains of NCSs, namely ICS and in-vehicle networks.

8.2 Industrial Control Systems

ICS is a term generally used to indicate several types of control systems (e.g., Supervisory Control And Data Acquisition or “SCADA”) used in industrial production

for monitoring and controlling physical processes. ICSs work over several domains such as energy, water treatment, manufacturing, etc. and embrace a wide family of technologies. Among them, Modbus [41], MMS [25], IEC104 [12], and DNP3 [10] are some of the most used protocols and standards deployed for industrial control.

Specification-based intrusion detection for ICSs is not new. Works such as [4] show the effectiveness of this approach applied to electrical grids. However, applying a set of specification rules to a real deployment still requires manually crafting all parameters on specific needs. Again, our research can improve the use of specification-based intrusion detection by leveraging available information of the deployments. For example, in the smart grid scenario, we would focus on Programmable Logic Controllers (PLCs) and Remote Terminal Units (RTUs), as these play a main role in the infrastructure. We would analyze variables handled by these controllers, their types, values and access methods and then use the abstract rules defined in §7.

Regarding the assumption of §6, verified information about the smart grid is *available* within configuration files that use the “Substation Configuration Language” (SCL). SCL files usually provide formal representations of modeled data and communication services. The information included in these files is *linkable* thanks to the included detail descriptions of the involved infrastructures (e.g., “Substation Configuration Description” files). Besides SCL files, operators usually store additional documentation describing physical and control processes as in the building automation use case. An IDS can leverage this documentation to gather further information and derive specification rules.

The three steps of the approach remain unchanged. System discovery will passively gather data about devices communicating over the ICS network. According to the verbosity of the involved protocols, an IDS will eventually collect enough information to identify infrastructure components and start the Feature Lookup step. Once information about PLCs and RTUs functioning is retrieved, Rule Definition will use it to define the actual specification rules.

8.3 In-Vehicle Networks

Similar argumentation can be applied to communication of Electronic Control Units (ECUs) over automotive bus systems like the “Controller Area Network” (CAN) or FlexRay found in all of today’s cars.

CAN is a network where connected ECUs communicate by means of small messages with a payload of only 8 bytes. CAN uses content-based addressing where messages only carry a 11 (or 21) bit message identifier, and receiving ECUs will select messages relevant to them

based on this message identifier. Message identifiers also serve as prioritization, as the employed CSMA/CR medium access scheme will always grant priority to the message with the lowest message identifier avoiding collisions on the bus. Transport layer protocols such as ISO-TP allow for transfer of longer messages fragmented into smaller network packets and more complex forms of addressing crossing gateways connecting multiple CAN segments.

In order to maintain and manage the assignment and semantics of message identifiers, the design phase of an automotive network involves setup of a so-called CAN-Matrix that lists exactly which ECU is supposed to sent which message identifier, which ECUs will receive messages of certain type and also the payload syntax and semantics. This design is done using sophisticated tools like Vector Informatics CANOE.⁵ The data provided by such tools is a perfect data source for specification-based IDS and for our approach, so the criteria of *availability* is met. *linkability* is more of a concern, as messages per se do not contain information on their source or type and a recipient needs to know (part of) the CAN matrix to identify how to decode a certain message ID. However, with the CAN matrix, we do have information on the types of ECUs available and can therefore conduct System discovery. This information can then be used to conduct Feature Lookup. A lot of relevant information (which messages are supposed to be seen on which bus segment) is again contained in the CAN matrix. Unfortunately, documentation in vehicular networks is not as standardized as the PICSs are in BACnet. So feature lookup would probably require more detailed investigations and more complex document parsing. Rule definition is then straightforward. However, having no source or destination addresses in packets, one would have to focus on message IDs, bus segments, and payload for detection.

While specification-based intrusion detection has been proposed many times especially for CAN-based networks [36, 31], a structured approach to rule-mining is missing in this domain so far and we see this as a promising field of application for our approach.

9 Conclusion

As networked control technologies are rapidly emerging, the need for securing these systems faces the key challenge of quickly scaling up to a multitude of heterogeneous devices. Our research aims to automate the deployment of effective security solutions, as well to adapt them in parallel with the monitored systems' lifecycle. More concretely, we present a novel approach to

specification-based intrusion detection for NCSs. While state-of-the-art solutions exploit manually-crafted specification rules, we discuss the feasibility of automatically mining these rules from available documentation. The tests performed on real building automation systems show the effectiveness of the obtained systems and confirm the time improvement in their development and deployment.

10 Acknowledgments

The authors would like to explicitly thank Dina Hadžiosmanović and Andreas Peter for the insightful discussions that gave rise to this research. Furthermore, the authors would like to acknowledge the work of Geert Jan Laanstra, Henk Hobbelink, Vincent Stoffer and Chris Weyandt at the University of Twente and the Lawrence Berkeley National Laboratory.

This research has been partially supported by the European Commission through project FP7-SEC-607093-PREEMPTIVE funded by the 7th Framework Program. This work has also been supported by the U.S. National Science Foundation under Award CNS-1314973. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators, and do not necessarily reflect the views of the sponsors.

References

- [1] ANSI/ASHRAE STANDARD 135-2012. A data communication protocol for building automation and control networks, 2012.
- [2] BACNET INTEREST GROUP EUROPE. Engineering data exchange template for BACnet systems - “description of the EDE data fields”, 2007.
- [3] BALEPIN, I., MALTSEV, S., ROWE, J., AND LEVITT, K. N. Using specification-based intrusion detection for automated response. In *Recent Advances in Intrusion Detection, 6th International Symposium, RAID 2003, Pittsburgh, PA, USA, September 8-10, Proceedings* (2003), pp. 136–154.
- [4] BERTHIER, R., AND SANDERS, W. H. Specification-based intrusion detection for advanced metering infrastructures. In *17th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2011, Pasadena, CA, USA, December 12-14* (2011), pp. 184–193.
- [5] CASELLI, M., HADŽIOSMANOVIĆ, D., ZAMBON, E., AND KARGL, F. On the feasibility of device fingerprinting in industrial control systems. In *Critical Information Infrastructures Security - 8th International Workshop, CRITIS 2013, Amsterdam, The Netherlands, September 16-18, Revised Selected Papers* (2013), pp. 155–166.
- [6] ČELEDA, P., KREJČÍ, R., AND KRMÍČEK, V. Flow-based security issue detection in building automation and control networks. In *Information and Communication Technologies - 18th EUNICE/IFIP WG 6.2, 6.6 International Conference, EUNICE 2012, Budapest, Hungary, August 29-31, Proceedings* (2012), pp. 64–75.

⁵http://vector.com/vi_canoe_en.html

- [7] CHAUGULE, A., XU, Z., AND ZHU, S. A specification based intrusion detection framework for mobile phones. In *Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, Proceedings* (2011), pp. 19–37.
- [8] CHEUNG, S., DUTERTRE, B., FONG, M., LINDQVIST, U., SKINNER, K., AND VALDES, A. Using model-based intrusion detection for SCADA networks. In *Proceedings of the SCADA Security Scientific Symposium, Miami Beach, Florida, USA, 7 December* (2007), pp. 1–12.
- [9] DENNING, D. E. An intrusion-detection model. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 7-9* (1986), pp. 118–133.
- [10] DIST-1815-WG. IEEE standard for electric power systems communications-distributed network protocol (DNP3), 2012. <https://standards.ieee.org/findstds/standard/1815-2012.html>.
- [11] ECHELON CORPORATION. LonTalk protocol specification v3.0, 1994. <http://www.enerlon.com/JobAids/Lontalk%20Protocol1%20Spec.pdf>.
- [12] EQUIPMENT, IEC TELECONTROL. Systems—part 5-104: Transmission protocols - network access for IEC 60870-5-101 using standard transport profiles.
- [13] FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. A sense of self for Unix processes. In *IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 6-8* (1996), pp. 120–128.
- [14] FOVINO, I. N., CARCANO, A., MUREL, T. D. L., TROMBETTA, A., AND MASERA, M. Modbus/DNP3 state-based intrusion detection system. In *24th IEEE International Conference on Advanced Information Networking and Applications, AINA 2010, Perth, Australia, April 20-13* (2010), pp. 729–736.
- [15] GILDEA, D., AND JURAFSKY, D. Automatic labeling of semantic roles. *Computational Linguistics* 28, 3 (2002), 245–288.
- [16] GILL, R., SMITH, J., AND CLARK, A. J. Specification-based intrusion detection in WLANs. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), Miami Beach, Florida, USA, 11-15 December* (2006), pp. 141–152.
- [17] GRANZER, W., KASTNER, W., NEUGSCHWANDTNER, G., AND PRAUS, F. Security in networked building automation systems. Tech. rep., 2005.
- [18] GRANZER, W., PRAUS, F., AND KASTNER, W. Security in building automation systems. *IEEE Trans. Industrial Electronics* 57, 11 (2010), 3622–3630.
- [19] GRÖNKVIST, J., HANSSON, A., AND SKÖLD, M. Evaluation of a specification-based intrusion detection system for AODV. In *The Sixth Annual Mediterranean Ad Hoc Networking Workshop* (2007), pp. 121–128.
- [20] GUPTA, R. A., AND CHOW, M. Networked control system: Overview and research trends. *IEEE Trans. Industrial Electronics* 57, 7 (2010), 2527–2535.
- [21] HADELÍ, H., SCHIERHOLZ, R., BRAENDLE, M., AND TUDUCE, C. Leveraging determinism in industrial control systems for advanced anomaly detection and reliable security configuration. In *Proceedings of 12th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2009, Palma de Mallorca, Spain, September 22-25* (2009), pp. 1–8.
- [22] HADŽIOSMANOVIĆ, D., BOLZONI, D., ETALLE, S., AND HARTEL, P. H. Challenges and opportunities in securing industrial control systems. In *Complexity in Engineering, COMPENG 2012, Aachen, Germany, June 11-13* (2012), pp. 1–6.
- [23] HASSAN, H. M., MAHMOUD, M., AND EL-KASSAS, S. Securing the AODV protocol using specification-based intrusion detection. In *Q2SWinet'06 - Proceedings of the Second ACM Workshop on QoS and Security for Wireless and Mobile Networks, Terramolinos, Spain, October 2* (2006), pp. 33–36.
- [24] HOLMBERG, D. G., AND EVANS, D. *BACnet Wide Area Network Security Threat Assessment*. US Department of Commerce, National Institute of Standards and Technology NIST, 2003.
- [25] ISO. Industrial automation systems – manufacturing message specification – part 2: Protocol specification, 2003.
- [26] JIEKE, P., REDOL, J., AND CORREIA, M. Specification-based intrusion detection system for carrier ethernet. In *WEBIST 2007 - Proceedings of the Third International Conference on Web Information Systems and Technologies, Volume IT, Barcelona, Spain, March 3-6* (2007), pp. 426–429.
- [27] JOKAR, P., NICANFAR, H., AND LEUNG, V. C. M. Specification-based intrusion detection for home area networks in smart grids. In *IEEE Second International Conference on Smart Grid Communications, SmartGridComm 2011, Brussels, Belgium, October 17-20* (2011), pp. 208–213.
- [28] KASTNER, W., NEUGSCHWANDTNER, G., SOUCEK, S., AND NEWMAN, M. H. Communication systems for building automation and control. *Proceedings of the IEEE* 93, 6 (2005), 1178–1203.
- [29] KAUR, J., TONEJC, J., WENDZEL, S., AND MEIER, M. Securing BACnet's pitfalls. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, Proceedings* (2015), pp. 616–629.
- [30] KIM, K., AND KUMAR, P. R. The importance, design and implementation of a middleware for networked control systems. *Springer Lecture Notes in Control and Information Sciences* 406, 1 (2010), 1–29.
- [31] KLEBERGER, P., OLOVSSON, T., AND JONSSON, E. Security aspects of the in-vehicle network in the connected car. In *IEEE Intelligent Vehicles Symposium (IV), 2011, Baden-Baden, Germany, June 5-9* (2011), pp. 528–533.
- [32] KNX ASSOCIATION. KNX Standard, 2011. <https://www.knx.org>.
- [33] KO, C., BRUTCH, P., ROWE, J., TSAFNAT, G., AND LEVITT, K. N. System health and intrusion monitoring using a hierarchy of constraints. In *Recent Advances in Intrusion Detection, 4th International Symposium, RAID 2001 Davis, CA, USA, October 10-12, Proceedings* (2001), pp. 190–204.
- [34] KO, C., FINK, G., AND LEVITT, K. N. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *10th Annual Computer Security Applications Conference, ACSAC 1994, Orlando, FL, USA, 5-9 December* (1994), pp. 134–144.
- [35] KO, C., RUSCHITZKA, M., AND LEVITT, K. N. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 4-7* (1997), pp. 175–187.
- [36] LARSON, U. E., NILSSON, D. K., AND JONSSON, E. An approach to specification-based attack detection for in-vehicle networks. In *IEEE Intelligent Vehicles Symposium (IV), 2008, Eindhoven, the Netherlands, June 4-6* (2008), pp. 220–225.
- [37] LIN, H., SLAGELL, A. J., MARTINO, C. D., KALBARTZYK, Z., AND IYER, R. K. Adapting Bro into SCADA: Building a specification-based intrusion detection system for the DNP3 protocol. In *Cyber Security and Information Intelligence, CSIRW '13, Oak Ridge, TN, USA, January 8-10* (2013), p. 5.

- [38] LYON, G. F. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009. <https://nmap.org/>.
- [39] MANYIKA, J., CHUI, M., BUGHIN, J., DOBBS, R., BISSON, P., AND MARRS, A. Disruptive technologies: Advances that will transform life, business, and the global economy. Tech. rep., 2013.
- [40] MATHERLY, J. C. SHODAN: the computer search engine, Jun 2016. <http://www.shodanhq.com/>.
- [41] MODBUS-IDA. Modbus application protocol specification v1.1b3, 2012. <http://www.modbus.org>.
- [42] NATIONAL JOINT APPRENTICESHIP & TECHNICAL COMMITTEE. *Building Automation: Control Devices and Applications*. American Technical Publishers, Inc., 2008.
- [43] NEWMAN, M. *BACnet: The Global Standard for Building Automation and Control Networks*. Momentum Press, 2013.
- [44] ORSET, J., ALCALDE, B., AND CAVALLI, A. R. An EFSM-based intrusion detection system for ad hoc networks. In *Automated Technology for Verification and Analysis, Third International Symposium, ATVA 2005, Taipei, Taiwan, October 4-7, Proceedings* (2005), pp. 400–413.
- [45] PAN, Z., HARIRI, S., AND AL-NASHIF, Y. B. Anomaly based intrusion detection for building automation and control networks. In *11th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2014, Doha, Qatar, November 10-13* (2014), pp. 72–77.
- [46] PAXSON, V. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29* (1998).
- [47] PEACOCK, M. D., AND JOHNSTONE, M. N. An analysis of security issues in building automation systems.
- [48] PETRONI, N. L., FRASER, T., WALTERS, A., AND ARBAUGH, W. A. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4* (2006).
- [49] SALSBURY, T. I. The smart building. In *Springer Handbook of Automation*. Springer, 2009, pp. 1079–1093.
- [50] SANEIFAR, H., BONNIOL, S., LAURENT, A., PONCELET, P., AND ROCHE, M. Terminology extraction from log files. In *Database and Expert Systems Applications, 20th International Conference, DEXA 2009, Linz, Austria, August 31 - September 4, Proceedings* (2009), pp. 769–776.
- [51] SANNER, M. F. Python: a programming language for software integration and development. *J Mol Graph Model* 17, 1 (1999), 57–61. <https://www.python.org/>.
- [52] SEKAR, R., CAI, Y., AND SEGAL, M. A specification-based approach for building survivable systems. In *Proceedings of the National Information Systems Security Conference (NISSC'98)* (1998), pp. 338–347.
- [53] SEKAR, R., GUPTA, A. K., FRULLO, J., SHANBHAG, T., TIWARI, A., YANG, H., AND ZHOU, S. Specification-based anomaly detection: A new approach for detecting network intrusions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22* (2002), pp. 265–274.
- [54] SEKAR, R., AND UPPULURI, P. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the 8th USENIX Security Symposium, Washington, D.C., August 23-26* (1999).
- [55] SOMMER, R., AMANN, J., AND HALL, S. Spicy: A unified deep packet inspection framework dissecting all your data. Tech. rep., ICSI, 2015. TR-15-004.
- [56] SONG, T., KO, C., TSENG, C. H., BALASUBRAMANYAM, P., CHAUDHARY, A., AND LEVITT, K. N. Formal reasoning about a specification-based intrusion detection for dynamic auto-configuration protocols in ad hoc networks. In *Formal Aspects in Security and Trust, Third International Workshop, FAST 2005, Newcastle upon Tyne, UK, July 18-19, Revised Selected Papers* (2005), pp. 16–33.
- [57] STRZALKOWSKI, T. Natural language information retrieval. *Inf. Process. Manage.* 31, 3 (1995), 397–417.
- [58] SZŁÓSARCZYK, S., WENDZEL, S., KAUR, J., MEIER, M., AND SCHUBERT, F. Towards suppressing attacks on and improving resilience of building automation systems - an approach exemplified using BACnet. In *Sicherheit 2014: Sicherheit, Schutz und Zuverlässigkeit, Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 19.-21. März 2014, Wien, Österreich* (2014), pp. 407–418.
- [59] TRUONG, P., NIEH, D., AND MOH, M. Specification-based intrusion detection for H.323-based voice over IP. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology (ISSPIT 2005), Athens, Greece, December 18-21* (2005), pp. 387–392.
- [60] TSENG, C., BALASUBRAMANYAM, P., KO, C., LIMPRASIT-TIPORN, R., ROWE, J., AND LEVITT, K. N. A specification-based intrusion detection system for AODV. In *Proceedings of the 1st ACM Workshop on Security of ad hoc and Sensor Networks, SASN 2003, Fairfax, Virginia, USA* (2003), pp. 125–134.
- [61] TSENG, C. H., SONG, T., BALASUBRAMANYAM, P., Ko, C., AND LEVITT, K. N. A specification-based intrusion detection model for OLSR. In *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, Revised Papers* (2005), pp. 330–350.
- [62] UPPULURI, P., AND SEKAR, R. Experiences with specification-based intrusion detection. In *Recent Advances in Intrusion Detection, 4th International Symposium, RAID 2001 Davis, CA, USA, October 10-12, Proceedings* (2001), pp. 172–189.
- [63] WENDZEL, S., KAHLER, B., AND RIST, T. Covert channels and their prevention in building automation protocols: A prototype exemplified using BACnet. In *2012 IEEE International Conference on Green Computing and Communications, Conference on Internet of Things, and Conference on Cyber, Physical and Social Computing, GreenCom/iThings/CPSCom 2012, Besancon, France, November 20-23* (2012), pp. 731–736.
- [64] ZALEWSKI, M. P0f: Passive OS fingerprinting tool, 2006. lcamtuf.coredump.cx/p0f3/.
- [65] ZHANG, P. *Industrial Control Technology: A Handbook for Engineers and Researchers*. William Andrew Inc., 2008.

Optimized Invariant Representation of Network Traffic for Detecting Unseen Malware Variants

Karel Bartos

Cisco Systems, Inc.

*Czech Technical University in Prague,
Faculty of Electrical Engineering*

Michal Sofka

Cisco Systems, Inc.

*Czech Technical University in Prague,
Faculty of Electrical Engineering*

Vojtech Franc

*Czech Technical University in Prague,
Faculty of Electrical Engineering*

Abstract

New and unseen polymorphic malware, zero-day attacks, or other types of advanced persistent threats are usually not detected by signature-based security devices, firewalls, or anti-viruses. *This represents a challenge to the network security industry as the amount and variability of incidents has been increasing.* Consequently, this complicates the design of learning-based detection systems relying on features extracted from network data. The problem is caused by different joint distribution of observation (features) and labels in the training and testing data sets. This paper proposes a classification system designed to detect both known as well as previously-unseen security threats. The classifiers use statistical feature representation computed from the network traffic and learn to recognize malicious behavior. The representation is designed and optimized to be invariant to the most common changes of malware behaviors. This is achieved in part by a feature histogram constructed for each group of HTTP flows (proxy log records) of a user visiting a particular hostname and in part by a feature self-similarity matrix computed for each group. The parameters of the representation (histogram bins) are optimized and learned based on the training samples along with the classifiers. The proposed classification system was deployed on large corporate networks, where it detected 2,090 new and unseen variants of malware samples with 90% precision (9 of 10 alerts were malicious), which is a considerable improvement when compared to the current flow-based approaches or existing signature-based web security devices.

1 Introduction

Current network security devices classify large amounts of the malicious network traffic and report the results in many individually-identified incidents, some of which are false alerts. On the other hand, a lot of malicious traf-

fic remains undetected due to the increasing variability of malware attacks. As a result, security analysts might miss severe complex attacks because the incidents are not correctly prioritized or reported.

The network traffic can be classified at different levels of detail. Approaches based on packet inspection and signature matching [15] rely on a database of known malware samples. These techniques are able to achieve results with high precision (low number of false alerts), but their detection ability is limited only to the known samples and patterns included in the database (limited recall). Moreover, due to the continuous improvements of network bandwidth, analyzing individual packets is becoming intractable on high-speed network links. It is more efficient to classify network traffic based on *flows* representing groups of packets (e.g. NetFlow [1] or proxy logs [26]). While this approach has typically lower precision, it uses statistical modeling and behavioral analysis [8] to find new and previously unseen malicious threats (higher recall).

Statistical features calculated from flows can be used for unsupervised anomaly detection, or in supervised classification to train data-driven classifiers of malicious traffic. While the former approach is typically used to detect new threats, it suffers from lower precision which limits its practical usefulness due to large amount of false alerts. Data-driven classifiers trained on known malicious samples achieve better efficacy results, but the results are directly dependent on the samples used in the training. Once a malware changes the behavior, the system needs to be retrained. With continuously rising number of malware variants, this becomes a major bottleneck in modern malware detection systems. Therefore, the robustness and invariance of features extracted from raw data plays the key role when classifying new malware.

The problem of changing malware behavior can be formalized by recognizing that a joint distribution of the malware samples (or features) differs for already known training (source) and yet unseen testing (target) data.

This can happen as a result of target evolving after the initial classifier or detector has been trained. In supervised learning, this problem is solved by domain adaptation. Under the assumption that the source and target distributions do not change arbitrarily, the goal of the domain adaptation is to leverage the knowledge in the source domain and transfer it to the target domain. In this work, we focus on the case where the conditional distribution of the observation given labels is different, also called a conditional shift.

The domain adaptation (or knowledge transfer) can be achieved by adapting the detector using importance weighting such that training instances from the source distribution match the target distribution [37]. Another approach is to transform the training instances to the domain of the testing data or to create a new data representation with the same joint distribution of observation and labels [4]. The challenging part is to design a meaningful transformation that transfers the knowledge from the source domain and improves the robustness of the detector on the target domain.

In this paper, we present a new optimized invariant representation of network traffic data that enables domain adaptation under conditional shift. The representation is computed for bags of samples, each of which consists of features computed from network traffic logs. The bags are constructed for each user and contain all network communication with a particular host-name/domain. The representation is designed to be invariant under shifting and scaling of the feature values and under permutation and size changes of the bags. This is achieved by combining bag histograms with an invariant self similarity matrix for each bag. All parameters of the representation are learned automatically for the training data using the proposed optimization approach.

The proposed invariant representation is applied to detect malicious HTTP traffic. We will show that the classifier trained on malware samples from one category can successfully detect new samples from a different category. This way, the knowledge of the malware behavior is correctly transferred to the new domain. Compared to the baseline flow-based representation or widely-used security device, the proposed approach shows considerable improvements and correctly classifies new types of network threats that were not part of the training data.

This paper has the following major contributions:

- **Classifying new malware categories** – we propose a supervised method that is able to detect new types of malware categories from a limited amount of training samples. Unlike classifying each category separately, which limits the robustness, we propose an invariant training from malware samples of multiple categories.

- **Bag representation of samples** – Instead of classifying flows individually, we propose to group flows into bags, where each bag contains flows that are related to each other (e.g. having the same user and target domain). Even though the concept of grouping flows together has been already introduced in the previously published work (e.g. in [32]), these approaches rely on a sequence of flow-based features rather than on more complex representation.
- **Features describing the dynamics of the samples** – To enforce the invariant properties of the representation, we propose to use a novel approach, where the features are derived from the self-similarity of flows within a bag. These features describe the dynamics of each bag and have many invariant properties that are useful when finding new malware variants and categories.
- **Learning the representation from the training data** – To optimize the parameters of the representation, we propose a novel method that combines the process of learning the representation with the process of learning the classifier. The resulting representation ensures easier separation of malicious and legitimate communication and at the same time controls the complexity of the classifier.
- **Large scale evaluation** – We evaluated the proposed representation on real network traffic of multiple companies. Unlike most of the previously published work, we performed the evaluation on highly imbalanced datasets as they appear in practice (considering the number of malicious samples), with most of the traffic being legitimate, to show the potential of the approach in practice. This makes the classification problem much harder. We provided a comparison with state-of-the-art approaches and a widely-used signature-based web security device to show the advantages of the proposed approach.

2 Related Work

Network perimeter can be secured by a large variety of network security devices and mechanisms, such as host-based or network-based Intrusion Detection Systems (IDS) [36]. We briefly review both systems, focusing our discussion on network-based IDS, which are the most relevant to the presented work.

Host-based IDS systems analyze malicious code and processes and system calls related to OS information. Traditional and widely-used anti-virus software or spyware scanners can be easily evaded by simple transformations of malware code. To address this weakness, methods of static analysis [30], [38] were proposed.

Static analysis, relying on semantic signatures, concentrates on pure investigation of code snippets without actually executing them. These methods are more resilient to changes in malware codes, however they can be easily evaded by obfuscation techniques. Methods of dynamic analysis [29], [34], [42] were proposed to deal with the weaknesses of static analysis, focusing on obtaining reliable information on execution of malicious programs. The downside of the dynamic analysis is the necessity to run the codes in a restricted environment which may influence malware behavior or difficulty of the analysis and tracing the problem back to the exact code location. Recently, a combination of static and dynamic analysis was used to analyze malicious browser extensions [20].

Network-based IDS systems are typically deployed on the key points of the network infrastructure and monitor incoming and outgoing network traffic by using static signature matching [15] or dynamic anomaly detection methods [8]. Signature-based IDS systems evaluate each network connection according to the predefined malware signatures regardless of the context. They are capable of detecting well-known attacks, but with limited amount of detected novel intrusions. On the other hand, anomaly-based IDS systems are designed to detect wide range of network anomalies including yet undiscovered attacks, but at the expense of higher false alarm rates [8].

Network-based approaches are designed to detect malicious communication by processing network packets or logs. An overview of the existing state-of-the-art approaches is shown in Table 1. The focus has been on the traffic classification from packet traces [5], [28], [39], [41], as this source provides detailed information about the underlying network communication. Due to the still increasing demands for larger bandwidth, analyzing individual packets is becoming intractable on high-speed network links. Moreover, some environments with highly confidential data transfers such as banks or government organizations do not allow deployment of packet inspection devices due to the legal or privacy reasons. The alternative approach is the classification based on network traffic logs, e.g. NetFlow [1], DNS records, or proxy logs. The logs are extracted at the transport layer and contain information only from packet headers.

Methods introduced in [12] and [23] apply features extracted from NetFlow data to classify network traffic into general classes, such as P2P, IMAP, FTP, POP3, DNS, IRC, etc. A comparison and evaluation of these approaches can be found in a comprehensive survey [24]. A combination of host-based statistics with SNORT rules to detect botnets was introduced in [16]. The authors showed that it is possible to detect malicious traffic using statistical features computed from NetFlow data, which motivated further research in this field. An alternative approach for classification of botnets from NetFlow fea-

tures was proposed in [6]. The authors of [33] have used normalized NetFlow features to cluster flow-based samples of network traffic into four predefined categories. As opposed to our approach, the normalization was performed to be able to compare individual features with each other. In our approach, we extended this idea and use normalization to be able to compare various malware categories. While all these approaches represent relevant state-of-the-art, network threats evolve so rapidly that these methods are becoming less effective due to the choice of features and the way they are used.

One of the largest changes in the network security landscape is the fact that HTTP(S) traffic is being used not only for web browsing, but also for other types of services and applications (TOR, multimedia streaming, remote desktop) including lots of malicious attacks. According to recent analysis [18], majority of malware samples communicate via HTTP. This change has drawn more attention to classifying malware from web traffic. In [25], the authors proposed an anomaly detection system composed of several techniques to detect attacks against web servers. They divide URIs into groups, where each group contains URIs with the same resource path. URIs without a query string or with return code outside of interval [200, 300] are considered as irrelevant. The system showed the ability to detect unseen malware samples and the recall will be compared with our proposed approach in Section 8. In [40], the authors introduced a method for predicting compromised websites using features extracted from page content and Alexa Web Information Service.

Having sufficient amount of labeled malware samples at disposal, numerous approaches proposed supervised learning methods to achieve better efficacy. Classifying DGA malware from DNS records based on connections to non-existent domains (NXDomains) was proposed in [2]. Even though several other data sources were used to detect malware (such as malware executions [3] or JavaScript analysis [22]), the most relevant work to our approach uses proxy logs [9], [17], [27], [44], [32].

In all these methods, proxy log features are extracted from real legitimate and malicious samples to train a data-driven classifier, which is used to find new malicious samples from the testing set. There are five core differences between these approaches and our approach: (1) we do not classify individual flows (in our case proxy log records), but sets of related flows called bags, (2) we propose a novel representation based on features describing the dynamics of each bag, (3) the features are computed from the bags and are invariant against various changes an attacker could implement to evade detection, (4) parameters of the proposed representation are learned automatically from the input data to maximize the detection performance, (5) the proposed classification system

Approach	Type	Method	Features	Target class	Testing Data			Malicious samples	Mal:All ratio
					Type	Year	All samples		
Wang [41]	U	anomaly detection	packet payload	worms, exploits	packets	2003	531,117	N/A	N/A
Kruegel [25]	U	anomaly detection	URL query parameters	web malware	proxy logs	2003	1,212,197	11	1:100k
Gu [16]	U	clustering	host statistics+SNORT	botnet	NetFlow	2007	100,000k	5,842k	1:17
Bilge [6]	S	random forest	flow size, time	botnets	NetFlow	2011	78,000,000	36	1:2.2M
Antonakakis [2]	S	multiple	NXDomains	dga malware	DNS data	2011	360,700	8008	1:45
Bailey [3]	S	hierarch. clustering	state changes	malware	executions	2007	4,591	4,591	1:1
Kapravelos [22]	S	similarity of trees	abstract syntax tree	web malware	JavaScript	2012	20,918,798	186,032	1:112
Choi [9]	S	SVM + RAKEL	URL lexical, host, dns	malicious flows	proxy logs	2009	72,000	32,000	1:2
Zhao [44]	S	active learning	URL lexical + host	malicious flows	proxy logs	2009	1,000,000	10,000	1:100
Huang [17]	S	SVM	URL lexical	phishing	proxy logs	2011	12,193	10,094	1:1
Ma [27]	S	multiple	URL lexical + host	malicious flows	proxy logs	2011	2,000,000	6,000	1:333
Invernizzi [18]	U	graph clustering	proxy log fields	mw downloads	proxy logs	2012	1,219	324	1:4
Soska [40]	S	random forests	content of web pages	infected websites	web pages	2014	386,018	49,347	1:8
Nelms [32]	S	heuristics	web paths	mw downloads	proxy logs	2014	N/A	150	N/A
Our approach	S	learned repr.+SVM	learned bag dynamics	malicious flows	proxy logs	2015	15,379,466	43,380	1:355

Table 1: Overview of the existing state-of-the-art approaches focusing on classification of malicious traffic (U = unsupervised, S = supervised). In contrast to the existing work, our approach proposes novel and optimized representation of bags, describing the dynamics of each legitimate or malicious sample. The approach is evaluated on latest real datasets with a realistic ratio of malicious and background flows (proxy log records).

was deployed on corporate networks and evaluated on imbalanced datasets (see Table 1) as they appear in practice to show the expected efficacy on these networks.

3 Formalization of the Problem

The paper deals with the problem of creating a robust representation of network communication that would be invariant against modifications an attacker can implement to evade the detection systems. The representation is used to classify network traffic into positive (malicious) or negative (legitimate) category. The labels for positive and negative samples are often very expensive to obtain. Moreover, sample distribution typically evolves in time, so the probability distribution of training data differs from the probability distribution of test data. This complicates the training of classifiers which assume that the distributions are the same. In the following, the problem is described in more detail.

Each sample is represented as an n -dimensional feature vector $x \in \mathbb{R}^n$. Samples are grouped into bags, with every bag represented as a matrix $X = (x_1, \dots, x_m) \in \mathbb{R}^{n \times m}$, where m is the number of samples in the bag and n is the number of features. The bags may have different number of samples. A single category y_i can be assigned to each bag from the set $\mathcal{Y} = \{y_1, \dots, y_N\}$. Only a few categories are included in the training set. The probability distribution on training and testing bags for category y_j will be denoted as $P^L(X|y_j)$ and $P^T(X|y_j)$, respectively. Moreover, the probability distribution of the training data differs from the probability distribution of the testing data, i.e. there is a domain adaptation problem [7] (also called a conditional shift [43]):

$$P^L(X|y_j) \neq P^T(X|y_j), \forall y_j \in \mathcal{Y}. \quad (1)$$

The purpose of the domain adaptation is to apply knowledge acquired from the training (source) domain into test (target) domain. The relation between $P^L(X|y_i)$ and $P^T(X|y_i)$ is not arbitrary, otherwise it would not be possible to transfer any knowledge. Therefore there is a transformation τ , which transforms the feature values of the bags onto a representation, in which $P^L(\tau(X)|y_i) \approx P^T(\tau(X)|y_i)$. The goal is to find this representation, allowing to classify individual bag represented as X into categories $\mathcal{Y} = \{y_1, \dots, y_N\}$ under the above mentioned conditional shift.

Numerous methods for transfer learning have been proposed (since the traditional machine learning methods cannot be used effectively in this case), including kernel mean matching [14], kernel learning approaches [11], maximum mean discrepancy [19], or boosting [10]. These methods try to solve a general data transfer with relaxed conditions on the similarity of the distributions during the transfer. The downside of these methods is the necessity to specify the target loss function and availability of large amount of labeled data.

This paper proposes an effective invariant representation that solves the classification problem with a covariate shift (see Equation 1). Once the data are transformed, the new feature values do not rely on the original distribution and they are not influenced by the shift. The parameters of the representation are learned automatically from the data together with the classifier as a joint optimization process. The advantage of this approach is that the parameters are optimally chosen during training to achieve the best classification efficacy for the given classifier, data, and representation.

4 Invariant Representation

The problem of domain adaptation outlined in the previous section is addressed by the proposed representation of bags. The new representation is calculated with a transformation that consists of three steps to ensure that the new representation will be invariant under scaling and shifting of the feature values and under permutation and size changes of the bags.

4.1 Scale Invariance

As stated in Section 3, the probability distribution of bags from the training set can be different from the test set. In the first step, the representation of bags is transformed to be invariant under scaling of the feature values. The traditional representation X of a bag that consists of a set of m samples $\{x_1, \dots, x_m\}$ can be written in a form of a matrix:

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ & \vdots & & \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{pmatrix}, \quad (2)$$

where x_{lk} denotes k -th feature value of l -th sample. This form of representation of samples and bags is widely used in the research community, as it is straightforward to use and easy to compute. It is a reasonable choice in many applications with a negligible shift in the source and target probability distributions. However, in the network security domain, the dynamics of the network environment causes changes in the feature values and the shift becomes more prominent. As a result, the performance of the classification algorithms using the traditional representation is decreased.

In the first step, the representation is improved by making the matrix X to be invariant under scaling of the feature values. **Scale invariance** guarantees that even if some original feature values of all samples in a bag are multiplied by a common factor, the values in the new representation remain unchanged. To guarantee the scale invariance, the matrix X is scaled locally onto the interval $[0, 1]$ as follows:

$$\tilde{X} = \begin{pmatrix} \tilde{x}_{11} & \dots & \tilde{x}_{1n} \\ \vdots & & \\ \tilde{x}_{m1} & \dots & \tilde{x}_{mn} \end{pmatrix} \quad \tilde{x}_{lk} = \frac{x_{lk} - \min_l(x_{lk})}{\max_l(x_{lk}) - \min_l(x_{lk})} \quad (3)$$

4.2 Shift Invariance

In the second step, the representation is transformed to be invariant against shifting. **Shift invariance** guarantees that even if some original feature values of all samples in a bag are increased/decreased by a given amount, the

values in the new representation remain unchanged. Let us define a *translation invariant distance function* $d : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ for which the following holds: $d(u, v) = d(u + a, v + a)$.

Let x_{pk}, x_{qk} be k -th feature values of p -th and q -th sample from bag matrix X . Then the distance between these two values will be denoted as $d(x_{pk}, x_{qk}) = s_{pq}^k$. The distance $d(x_{pk}, x_{qk})$ is computed for pairs of k -th feature value for all sample pairs, ultimately forming a so called self-similarity matrix S^k . Self-similarity matrix is a symmetric positive semidefinite matrix, where rows and columns represent individual samples and (i, j) -th element corresponds to the distance between i -th and j -th sample. Self-similarity matrix has been already used thanks to its properties in several applications (e.g. in object recognition [21] or music recording [31]). However, only a single self-similarity matrix for each bag has been used in these approaches. This paper proposes to compute a set of similarity matrices, one for every feature. More specifically, a per-feature set of self-similarity matrices $\mathcal{S} = \{S^1, S^2, \dots, S^n\}$ is computed for each bag, where

$$S^k = \begin{pmatrix} s_{11}^k & s_{12}^k & \dots & s_{1m}^k \\ & \vdots & & \\ s_{m1}^k & s_{m2}^k & \dots & s_{mm}^k \end{pmatrix}. \quad (4)$$

The element $s_{pq}^k = d(x_{pk}, x_{qk})$ is a distance between feature values x_{pk} and x_{qk} of k -th feature. This means that the bag matrix X with m samples and n features will be represented with n self-similarity matrices of size $m \times m$. The matrices are further normalized by local feature scaling described in Section 4.1 to produce a set of matrices $\tilde{\mathcal{S}}$.

The shift invariance makes the representation robust to the changes where the feature values are modified by adding or subtracting a fixed value. For example, the length of a malicious URL would change by including an additional subdirectory in the URL path. Or, the number of transferred bytes would increase when an additional data structure is included in the communication exchange.

4.3 Permutation and Size Invariance

Representing bags with scaled matrices $\{\tilde{X}\}$ and sets of locally-scaled self-similarity matrices $\{\tilde{\mathcal{S}}\}$ achieves the scale and shift invariance. **Size invariance** ensures that the representation is invariant against the size of the bag. In highly dynamic environments, the samples may occur in a variable ordering. **Permutation invariance** ensures that the representation should also be invariant against any reordering of rows and columns of the matrices. The final step of the proposed transformation is the transition from the scaled matrices $\tilde{X}, \tilde{\mathcal{S}}$ (introduced in Sec-

tions 4.1 and 4.2 respectively) to normalized histograms. For this purpose, we define for each bag:

$z_k^X :=$ vector of values from k -th column of matrix \tilde{X}

$z_k^S :=$ column-wise representation of upper triangular matrix created from matrix $\tilde{S}^k \in \mathcal{S}$.

This means that $z_k^X \in \mathbb{R}^m$ is a vector created from values of k -th feature of \tilde{X} , while $z_k^S \in \mathbb{R}^r, r = (m-1) \cdot \frac{m}{2}$ is a vector that consists of all values of upper triangular matrix created from matrix \tilde{S}^k . Since \tilde{S}^k is a symmetric matrix with zeros along the main diagonal, z_k^S contains only values from upper triangular matrix of \tilde{S}^k .

A normalized histogram of vector $z = (z_1, \dots, z_d) \in \mathbb{R}^d$ is a function $\phi: \mathbb{R}^d \times \mathbb{R}^{b+1} \rightarrow \mathbb{R}^b$ parametrized by edges of b bins $\theta = (\theta_0, \dots, \theta_b) \in \mathbb{R}^{b+1}$ such that $\phi(z; \theta) = (\phi(z; \theta_0, \theta_1), \dots, \phi(z; \theta_{b-1}, \theta_b))$ where

$$\phi(z, \theta_i, \theta_{i+1}) = \frac{1}{d} \sum_{j=1}^d \llbracket z_j \in [\theta_{i-1}, \theta_i] \rrbracket$$

is the value of the i -th bin corresponding to a portion of components of z falling to the interval $[\theta_{i-1}, \theta_i]$.

Each column k of matrix \tilde{X} (i.e. all bag values of k -th feature) is transformed into a histogram $\phi(z_k^X, \theta_k^X)$ with predefined number of b bins and θ_k^X bin edges. Such histograms created from the columns of matrix \tilde{X} will be denoted as *feature values histograms*, because they carry information about the distribution of bag feature values. On the other hand, histogram $\phi(z_k^S, \theta_k^S)$ created from values of self-similarity matrix $\tilde{S}^k \in \mathcal{S}$ will be called *feature differences histograms*, as they capture inner feature variability within bag samples.

Overall, each bag is represented as a concatenated feature map $\phi(\tilde{X}; \mathcal{S}; \theta): \mathbb{R}^{n \times (m+r)} \rightarrow \mathbb{R}^{2 \cdot n \cdot b}$ as follows:

$$(\phi(z_1^X, \theta_1^X), \dots, \phi(z_n^X, \theta_n^X), \phi(z_1^S, \theta_1^S), \dots, \phi(z_n^S, \theta_n^S)) \quad (5)$$

where n is the number of the original flow-based features, m is the number of flows in the bag, and b is the number of bins. The whole transformation from input network flows to the final feature vector is depicted in Figure 1. As you can see, two types of invariant histograms are created from values of each flow-based feature. At the end, both histograms are concatenated into the final bag representation $\phi(\tilde{X}; \mathcal{S}; \theta)$.

5 Learning Optimal Histogram Representation

The bag representation $\phi(\tilde{X}; \mathcal{S}; \theta)$ proposed in Section 4 has the invariant properties, however it heavily depends on the number of bins b and their edges θ defining the

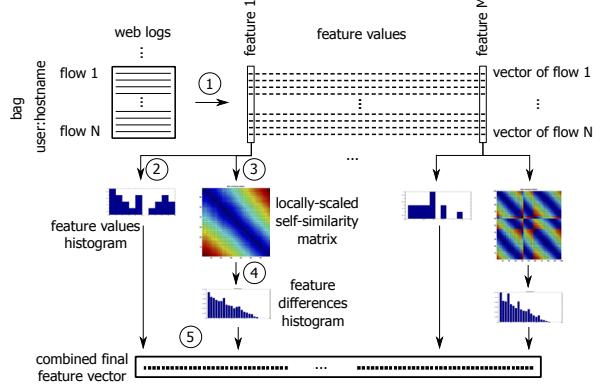


Figure 1: Graphical illustration of the individual steps that are needed to transform the bag (set of flows with the same user and hostname) into the proposed invariant representation. First, the bag is represented with a standard feature vector (1). Then feature values histograms of locally scaled feature values are computed for each feature separately (2). Next, the locally-scaled self-similarity matrix is computed for each feature (3) to capture inner differences. This matrix is then transformed into feature differences histogram (4), which is invariant on the number or the ordering of the samples within the bag. Finally, feature values and feature differences histograms of all features are concatenated into resulting feature vector.

width of the histogram bins. These parameters that were manually predefined in Section 4 C influence the classification performance. Incorrectly chosen parameters b and θ leads to suboptimal efficacy results. To define the parameters optimally, we propose a novel approach of learning these parameters automatically from the training data in such a way to maximize the classification separability between positive and negative samples.

When creating histograms in Section 4 C, the input instances are vectors z_k^X and z_k^S , where $k \in \{1, \dots, n\}$. The algorithm transforms the input instances into a concatenated histogram $\phi(\tilde{X}; \mathcal{S}; \theta)$. To keep the notation simple and concise, we will denote the input instances simply as $z = (z_1, \dots, z_n) \in \mathbb{R}^{n \times m}$ (instead of $z = (z_1^X, \dots, z_n^X, z_1^S, \dots, z_n^S)$), which is a sequence of n vectors each of dimension m .

The input instance z is represented via a feature map $\phi: \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \cdot b}$ defined as a concatenation of the normalized histograms of all vectors in that sequence, that is, $\phi(z; \theta) = (\phi(z_1; \theta_1), \dots, \phi(z_n; \theta_n))$, where $\theta = (\theta_1, \dots, \theta_n)$ denotes bin edges of all normalized histograms stacked to a single vector.

We aim at designing a classifier $h: \mathbb{R}^{n \times m} \times \mathbb{R}^{n+1} \times \mathbb{R}^{n(b+1)} \rightarrow \{-1, +1\}$ working on top of the histogram representation, that is

$$h(z; w, w_0, \theta) = \text{sign}(\langle \phi(z, w) \rangle + w_0) \\ = \text{sign} \left(\sum_{i=1}^n \sum_{j=1}^b \phi(z_i, \theta_{i,j-1}, \theta_{i,j}) w_{i,j} + w_0 \right). \quad (6)$$

The classifier (6) is linear in the parameters (w, w_0) but non-linear in θ and z . We are going to show how to learn parameters (w, w_0) and implicitly also θ via a convex optimization.

Assume we are given a training set of examples $\{(z^1, y^1), \dots, (z^m, y^m)\} \in (\mathbb{R}^{n \times m} \times \{+1, -1\})^m$. We fix the representation ϕ such that the number of bins b is sufficiently large and the bin edges θ are equally spaced. We find the weights (w, w_0) by solving

$$\min_{w \in \mathbb{R}^{b \cdot p}, w_0 \in \mathbb{R}} \left[\gamma \sum_{i=1}^n \sum_{j=1}^{b-1} |w_{i,j} - w_{i,j+1}| \right. \\ \left. + \frac{1}{m} \sum_{i=1}^m \max \{0, 1 - y^i \langle \phi(z^i; \theta), w \rangle\} \right]. \quad (7)$$

The objective is a sum of two convex terms. The second term is the standard hinge-loss surrogate of the training classification error. The first term is a regularization encouraging weights of neighboring bins to be similar. If it happens that j -th and $j+1$ bin of the i -the histogram have the same weight, $w_{i,j} = w_{i,j+1} = w$, then these bins can be effectively merged to a single bin because

$$w_{i,j} \phi(z_i; \theta_{i,j-1}, \theta_{i,j}) + w_{i,j+1} \phi(z_i; \theta_{i,j}, \theta_{i,j+1}) \\ = 2w \phi(z_i; \theta_{i,j-1}, \theta_{i,j+1}). \quad (8)$$

The trade-off constant $\gamma > 0$ can be used to control the number of merged bins. A large value of γ will result in massive merging and consequently in a small number of resulting bins. Hence the objective of the problem (7) is to minimize the training error and to simultaneously control the number of resulting bins. The number of bins influences the expressive power of the classifier and thus also the generalization of the classifier. The optimal setting of λ is found by tuning its value on a validation set.

Once the problem (7) is solved, we use the resulting weights w^* to construct a new set of bin edges θ^* such that we merge the original bins if the neighboring weights have the same sign (i.e. if $w_{i,j}^* w_{i,j+1}^* > 0$). This implies that the new bin edges θ^* are a subset of the original bin edges θ , however, their number can be significantly reduced (depending on γ) and they have different widths unlike the original bins. Having the new bins defined, we learn a new set of weights by the standard SVM algorithm

$$\min_{w \in \mathbb{R}^n, w_0 \in \mathbb{R}} \left[\frac{\lambda}{2} \|w\|^2 + \frac{1}{m} \sum_{i=1}^m \max \{0, 1 - y^i \langle \phi(z^i; \theta^*), w \rangle\} \right].$$

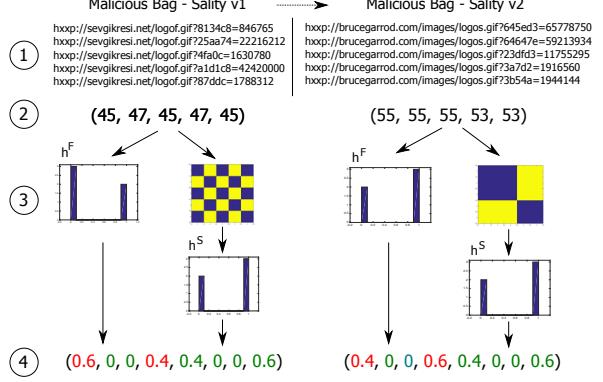


Figure 2: Illustration of the proposed representation applied on two versions of malware Sality. First, two bags of flows are created (1), one bag for each Sality sample. Next, flow-based feature vectors are created for each bag (2). For illustrative purposes, only a single feature is used - URL length. In the third step, histograms of feature values $\phi(z_k^X, \theta_k^X)$ and feature differences $\phi(z_k^S, \theta_k^S)$ are created (3) as described in Section 4.3. Only four bins for each histogram were used. Finally, all histograms are concatenated into the final feature vector (4). Even though the malware samples are from two different versions, they have the same histogram of feature differences $\phi(z_k^S, \theta_k^S)$. Since $\phi(z_k^X, \theta_k^X)$ is not invariant against shift, you can see that half of the values of $\phi(z_k^X, \theta_k^X)$ are different. Still, $\phi(z_k^X, \theta_k^X)$ values may play an important role when separating malware samples from other legitimate traffic.

Note that we could add the quadratic regularizer $\frac{\lambda}{2} \|w\|^2$ to the objective of (7) and learn the weights and the representation in a single stage. However, this would require tuning two regularization parameters (λ and γ) simultaneously which would be order of magnitude more expensive than tuning them separately in the two stage approach.

6 Malware Representation Example

This Section illustrates how the proposed representation (nonoptimized version) is calculated for two real-world examples of malicious behavior. Namely, two versions of a polymorphic malware Sality are compared. Sality [13] is a malware family that has become a dynamic and complex form of malicious infection. It utilizes polymorphic techniques to infect files of Windows machines. Signature-based systems or classifiers trained on a specific malware type often struggle with detecting new variants of this kind of malware. Note that most of the conclusions to the discussion that follows can be drawn for many other malware threats.

Figure 2 shows how the two Sality samples are represented with the proposed approach. First, the input flows are grouped into two bags (one bag for each Sality sample), because all flows of each bag have the same user and the same hostname (1). For the sake of simplicity, only URLs of the corresponding flows are displayed. Next, 88 flow-based feature vectors are computed for each bag (2). To simplify illustration, we use only a single feature – URL length. After this step, each Sality sample is represented with one feature vector of flow-based values. Existing approaches use these vectors as the input for the subsequent detection methods. As we will show in Section 7, these feature values are highly variable for malware categories. Classification models trained with such feature values loose generalization capability.

To enhance the robustness of the flow-based features, the proposed approach computes histograms of feature values $\phi(z_k^X, \theta_k^X)$ and feature differences $\phi(z_k^S, \theta_k^S)$ (3) as described in Section 4.3. To make the illustration simple, only four bins for each histogram were used. Finally, all histograms are concatenated into the final feature vector (4). It can be seen that even though the malware samples are from two different versions, they have the same histogram of feature differences $\phi(z_k^S, \theta_k^S)$. Since the histogram of feature values $\phi(z_k^X, \theta_k^X)$ is not invariant against shift, half of the values of $\phi(z_k^X, \theta_k^X)$ are different.

The number of histogram bins and their sizes are then learned from the data by the proposed algorithm (see Section 5). The proposed representation describes inner dynamics of flows from each bag, which is a robust indicator of malware samples, as we will show in the analysis of various malware families in Section 8. In contrast to the existing methods that use flow-based features or general statistics such as mean or standard deviation, the proposed representation reflects properties that are much more difficult for an attacker to evade detection.

7 Evasion Possibilities

This section discusses evasion options for an attacker when trying to evade a learning-based classification system. According to the recent work [35], the essential components for an evasion are: (1) the set of features used by the classifier, (2) the training dataset used for training, (3) the classification algorithm with its parameters. Without the knowledge of the features, the attacker is faced with major challenges and there is not any known technique for addressing them [35].

Acquire knowledge of classification algorithm with its parameters or the training data is hard if not impossible. Therefore, in the following analysis, we assume that only the features are known to the attacker. When classifying HTTP traffic from proxy logs, it is actually not difficult to create a set of common features widely used

in practice. These features are the baseline flow-based features, such as those described in Table 3. When the attacker performs a mimicry attack, selected features of malicious flows are modified to mimic legitimate traffic (or flows marked as benign by the classifier).

In the following, we will analyze the case when the attacker performs a mimicry attack to evade detection by modifying flow attributes, such as URLs, bytes, and inter-arrival times. Other flow attributes can be altered in a similar way with analogical results. All modifications are divided into two groups, depending on whether the proposed representation is invariant against them.

The proposed representation is invariant to the following changes.

- **Malicious code, payload, or obfuscation** – The advantage of all network-based security approaches is that they extract features from headers of network communication rather than from the content. As a result, any changes to the payload including the usage of pluggable transports designed to bypass Deep Packet Inspection (DPI) devices will have no effect on the features. Some pluggable transports (e.g. ScrambleSuit) are able to change its network fingerprint (packet length distribution, number of bytes, inter-arrival times, etc.). Since the proposed representation mainly relies on the dynamics of URLs of flows in the bag, such changes will not negatively impact the efficacy, which is a great advantage against DPI devices.
- **Server or hostname** – The representation operates at the level of bags, where each bag is a set of flows with the same user and hostname/domain. If an attacker changes an IP address or a hostname of the remote server (because the current one has been blacklisted), the representation will create a new bag with similar feature values as in the previous bag with the original IP address or hostname, which is a great advantage against feeds and blacklists that need to be updated daily and are always behind.
- **URL path or filename** – Straightforward and easy way of evading existing classifiers using flow-based features or URL patterns is the change in path or filename from sample to sample. Since the variability of these features remains constant within each bag, these changes will also have no effect on the proposed representation.
- **Number of URL parameters, their names or values** – This is an alternative to URL path changes.
- **Encoded URL content** – Hiding information in the URL string represents another way to exfiltrate sensitive data. When the URL is encrypted and encoded (e.g. with base64), it changes the URL length

and may globally influence other features as well. As the proposed representation is invariant against shifting, changing the URL length will not change the histograms of feature differences.

- **Number of flows** – Another option for an attacker to hide in the background traffic is increasing or reducing the number of flows related to the attack. Such modification of the attack does not affect the representation, as long as there are enough flows to create the feature vectors.
- **Time intervals between flows** – This feature has been used in many previous approaches for its descriptive properties. It is an alternative way to the proposed representation how to model a relationship between individual flows. Our analysis revealed that current malware samples frequently modify the inter-arrival time to remain hidden in the background traffic – see Figure 3 for details. Therefore, we do not rely on this unstable feature that can be also influenced by network delays or failures.
- **Ordering of flows** – An attacker can easily change the ordering of flows to evade detection based on patterns or predefined sequences of flows. For the proposed representation the ordering of flows does not matter.

The proposed representation is not invariant to the following changes.

- **Static behavior** – The representation does not model malware behaviors, where all flows associated with a malware are identical. Such behavior has no dynamics and can be classified with flow-based approaches with comparable results. In our dataset, only 10% of flows were removed because of this constraint.
- **Multiple behaviors in a bag** – In case more behaviors are associated with a bag, such as when a target hostname is compromised and communicates with a user with legitimate and malicious flows at once, the representation does not guarantee the invariance against the attacker’s changes. Such bags contain a mixture of legitimate and malicious flows and their combination could lead to a different representation. Note that there wasn’t any malware sample in our data that would satisfy this condition, since the legitimate traffic has to be authentic (not artificially injected) to confuse the representation.
- **Encrypted HTTPS traffic** – Most features presented in this paper are computed from URLs or other flow fields, that are not available in encrypted HTTPS traffic. In this case, only a limited set

Category	Samples		Signatures
	Flows	Bags	Recall
Training Positives	132,756	5,011	0.15
Click-fraud mw	12,091	819	0.29
DGA malware	8,629	397	0.58
Dridex	8,402	264	0.12
IntallCore	17,317	1,332	0.00
Monetization	3,107	135	0.00
Mudrop	37,142	701	0.00
Poweliks	11,648	132	0.00
Zeus	34,420	1,275	0.19
Testing Positives	43,380	2,090	0.02
Training Negatives	862,478	26,825	
Testing Negatives	15,379,466	240,549	

Table 2: Number of flows and bags of malware categories and legitimate background traffic used for training and testing the proposed representation and classifier. Right-most column shows the amount of bags that were found and blocked by an existing signature-based device. Majority of the malicious bags from the test were missed, as the device, relying on a static database of signatures, was not able to catch evolving versions and new types of the malicious behaviors.

of flow-based features can be used, which reduces the discriminative properties of the representation. However, majority of malware communication is still over HTTP protocol, because switching to HTTPS would harm the cyber-criminals’ revenues due to problems with signed certificates [18].

- **Real-time changes and evolution** – In case a malware sample for a given user and hostname would start changing its behavior dynamically and frequently, the bag representation will vary in time. Such inconsistency would decrease the efficacy results and enlarge the time to detect. However, creating such highly dynamic malware behavior requires a considerable effort, therefore we do not see such samples very often in the real network traffic.

We conclude our analysis with the observation, that attackers change flow features very frequently (see Figure 3). The goal of the proposed representation is to be invariant against most of the changes to successfully detect new, previously unseen malware variants.

8 Experimental Evaluation

The proposed approach was deployed on the top of proxy logs exporters in companies of various types and sizes to detect unseen malware samples. The system architecture is shown in Figure 4. Collector connected to a

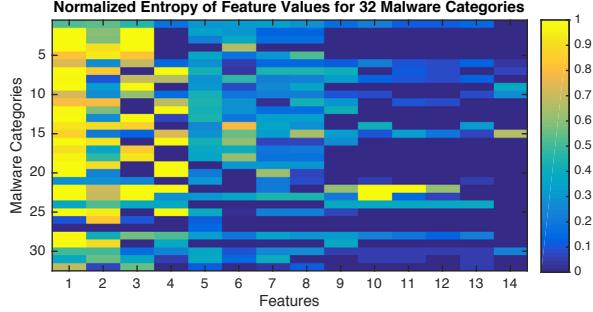


Figure 3: Flow-based features (columns) are changing for most of the malware categories (rows). The figure uses normalized entropy to show the variability of each feature within each malware category. Yellow color denotes that the feature value is changed very often, while blue color means that the feature has the same values for all samples of the given category. **Features:** 1-URL, 2-interarrival time, 3-URL query values, 4-URL path, 5-number of flows, 6-number of downloaded bytes, 7-server IP address, 8-hostname, 9-URL path length, 10-URL query names, 11-filename, 12-filename length, 13-number of URL query parameters, 14-number of uploaded bytes. **Malware categories:** 1-Click-fraud (amz), 2-Asterope family 1, 3-Asterope family 2, 4-Beden, 5-Click-fraud, 6-DGA, 7-Dridex, 8-Exfiltration, 9-InstallCore, 10-Mudrop Trojan Dropper, 11-Monetization, 12-Zeus, 13-Mudrop, 14-MultiPlug, 15-mixture of unknown malware, 16-Click-fraud (tracking), 17-Poweliks family 1, 18-Poweliks family 2, 19-Qakbot Trojan, 20-Rerdom Trojan, 21-Ramnit worm, 22-RVX, 23-Sality, 24-Threats related to a traffic direction system (TDS) 1, 25-TDS 2, 26-TDS 3, 27-Tinba Trojan, 28-C&C tunneling, 29-Uptare, 30-Vawtrak, 31-Vittalia, 32-Zbot. Details about the malware categories are given in Section 8.

proxy server stores incoming and outgoing network traffic in form of proxy log records. The proxy logs represent information about individual HTTP/HTTPS connections or flows. Each 5-minute interval, the proxy logs are sent to the detection engine, where the proposed method detects the malicious behaviors. Report created from the malicious behaviors is then displayed on a console to an operator. The next section provides the specification of datasets and malware categories, followed by the results from the experimental evaluation. Next section provides the specification of datasets and malware categories, followed by the results from the experimental evaluation.

8.1 Specification of the Datasets

The data was obtained from several months (January - July 2015) of real network traffic of 80 international

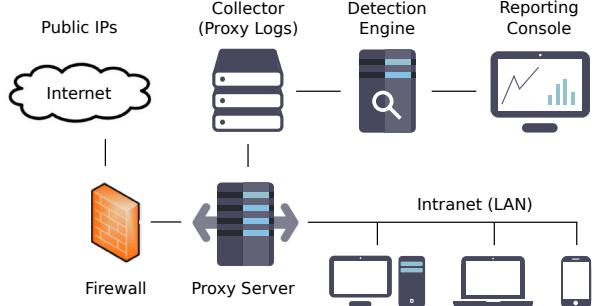


Figure 4: Overview of the system architecture. Collector connected to a proxy server stores incoming and outgoing network traffic in form of proxy log records. Each 5-minute interval, the proxy logs are sent to the detection engine and the results are displayed to an operator on the reporting console.

companies of various sizes in form of proxy logs [26]. The logs contain HTTP/HTTPS flows, where one flow is one connection defined as a group of packets from a single host and source port with a single server IP address, port, and protocol. Summary of the datasets used in the evaluation is described in Table 2.

Malware samples will be referred as **positive bags**, where one positive bag is a set of records (connections) with the same source towards the same destination. The bags not labeled as malicious are considered as legitimate/negative. Each bag should contain at least 5 flows to be able to compute a meaningful histogram representation. Training dataset contains 5k malicious (8 malware families) and 27k legitimate bags, while testing dataset is consist of 2k malicious (\gg 32 malware families) and 241k legitimate bags (more than 15 million flows). Positive samples for training were acquired using many types of publicly available feeds, services, and blacklists, while the results on the testing data were analyzed manually by security experts. Each HTTP flow consists of the following fields: user name, srcIP, dstIP, srcPort, dstPort, protocol, number of bytes, duration, timestamp, user agent, and URL. From these flow fields, we extracted 115 flow-based features typically used in the prior art (Table 3).

This means that training and testing data are composed of completely different malware bags from different malware families, which makes the classification problem much harder. This scenario simulates the fact that new types of threats are created to evade detection. The benchmarking signature-based network security device (widely used in many companies) was able to detect only 2% of the malicious bags from the testing set. Training a classifier for each category separately is an easier task, however such classifiers are typically overfitted to a single category and cannot detect further variations without retraining.

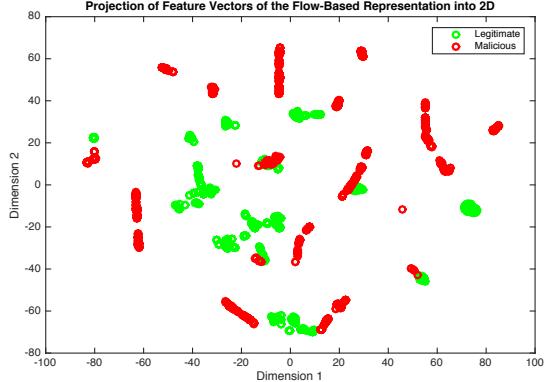


Figure 5: Graphical projection of feature vectors of the baseline flow-based representation into two dimensions using t-SNE transformation. Feature vectors from 32 different malware categories are displayed. Due to high variability of flow-based feature values, legitimate and malicious samples are scattered without any clear separation. The results show that the flow-based representation is suitable for training classifiers specialized on a single malware category, which often leads to classifiers with high precision and low recall.

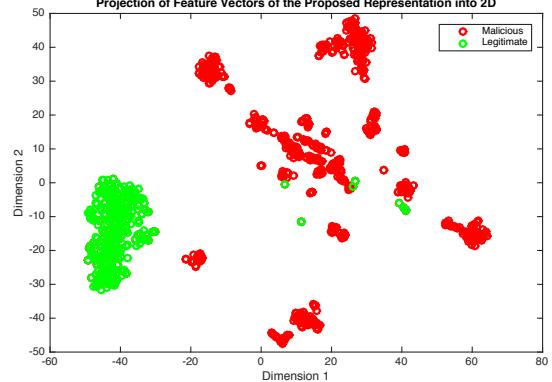


Figure 6: Graphical projection of feature vectors of the proposed representation into two dimensions using t-SNE transformation. Thanks to the invariant properties, malicious bags from various categories are grouped together, as they have similar dynamics modeled by the representation. Most of the legitimate bags are concentrated on the left-hand side, far from the malicious bags. This shows that training a classifier with the proposed representation will achieve higher recall with comparable precision.

Features applied on URL, path, query, filename
length; digit ratio
lower/upper case ratio; ratio of digits
vowel changes ratio
ratio of a character with max occurrence
has a special character
max length of consonant/vowel/digit stream
number of non-base64 characters
has repetition of parameters
Other Features
number of bytes from client to server
number of bytes from server to client
length of referer/file extension
number of parameters in query
number of '/' in path/query/referer

Table 3: List of selected flow-based features extracted from proxy logs. We consider these features as baseline (as some features were used in previously published work), and compare it with the proposed representation.

Table 4 from Appendix A describes an important fact about the URLs from individual malicious bags. As you can see, URLs within each malicious bag are similar to each other (as opposed to most of legitimate bags). This small non-zero variability of flow-based feature values is captured by the proposed representation using both types of histograms. The variability is very general but also

descriptive feature, which increases the robustness of the representation to further malware changes and variants.

8.2 Evaluation on Real Network Traffic

This section shows the benefits of the proposed approach of learning the invariant representation for two-class classification problem in network security. Feature vectors described in Section 8.1 correspond to input feature vectors $\{x_1, \dots, x_m\}$ defined in Section 3. These vectors are transformed into the proposed representation of histograms $\phi(\hat{X}; \mathcal{P}; \theta)$, as described in Section 4. We have evaluated two types of invariant representations. One with predefined number of equidistant bins (e.g. 16, 32, etc.) computed as described in Section 4, and one when the representation is learned together with the classifier to maximize the separability between malicious and legitimate traffic (combination of Section 4 and 5). For the representation learning, we used 256 bins as initial (and most detailed) partitioning of the histograms. During the learning phase, the bins were merged together, creating 12.7 bins per histogram on average.

Both approaches are compared with the baseline flow-based representation used in previously published work, where each sample corresponds to a feature vector computed from one flow. Results of a widely used signature-based security device are also provided (see Table 2) to demonstrate that the positive samples included in the evaluation pose a real security risk, as majority of them

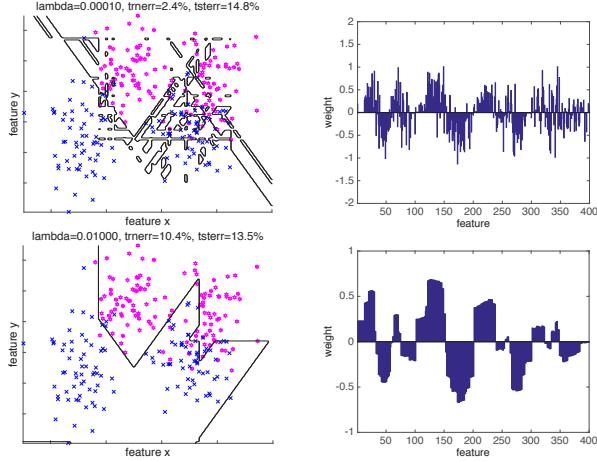


Figure 7: Visualization of the proposed method of learning the invariant representation on 2-dimensional synthetic data. Figures in the left row show the decision boundaries of two class classifier learned from the bins for two different values of parameter λ (0.0001, 0.01) which controls the number of emerging bins (the corresponding weights are shown in the right row). With increasing λ the data are represented with less bins and the boundary becomes smoother and less over-fitted to the training data.

was not detected. Maximum number of flows for each bag was 100, which ensures that the computational cost is controlled and does not exceed predefined limits.

Two-dimensional projection of the feature vectors for the flow-based and the proposed representation is illustrated in Figures 5 and 6 respectively. Bags from 32 malicious categories are displayed with red circles, while the legitimate bags are denoted with green circles. The projections show that the flow-based representation is suitable for training classifiers specialized on a single malware category. In case of the proposed representation, malicious bags from various categories are grouped together and far from the legitimate traffic, which means that the classifiers will have higher recall and comparable precision with the flow-based classifiers.

Next, we will show the properties of the proposed method of learning the representation to maximize the separation between positive and negative samples (see Section 5 for details). Figure 7 visualizes the proposed method on synthetic 2-dimensional input data. The input 2D point $(x, y) \in \mathbb{R}^2$ is represented by 4-dimensional feature vector $(x^2, y^2, x + y, x - y)$. Each of the 4 features is then represented by a histogram with 100 bins (i.e. each feature is represented by 100 dimensional binary vector with all zeros but a single one corresponding to the active bin). Figures in the top row show the decision boundaries of two-class classifiers learned from data. The bot-

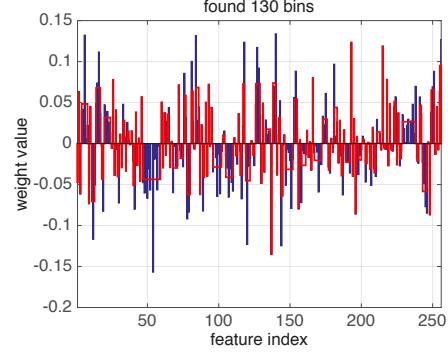


Figure 8: Weights (blue bars) and derived bins of a histogram (red line) for a standard SVM and one of the invariant features. Since the bins are equidistant and pre-defined at the beginning, the resulting histogram (defined by the red line) has complicated structure, leading most probably to complex boundary and over-fitted results (as shown in Figure 7 on the left hand side).

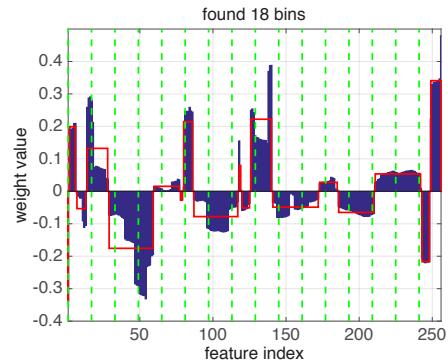


Figure 9: Weights (blue bars) and derived bins of a histogram (red line) for the proposed bin optimization. In this case, the weights show a clear structure and the derived histogram has only 18 bins. The decision boundary is in this case smoother and the classifier trained from this representation will be more robust. Green dashed lines also show how the histogram bins would look like if they are positioned equidistantly (16 bins).

tom row shows the weights of the linear classifier corresponding to the bins (in total 400 weights resulting from 100 bins for each out of 4 features). The columns correspond to the results obtained for different setting of the parameter λ which controls the number of emerging bins and thus also the complexity of the decision boundary. With increasing λ the data are represented with less bins and the boundary becomes smoother. Figure 7 shows the principle of the proposed optimization process. The bins of the representation are learned in such a way that it is much easier for the classifier to separate negative and positive samples and at the same time control the com-

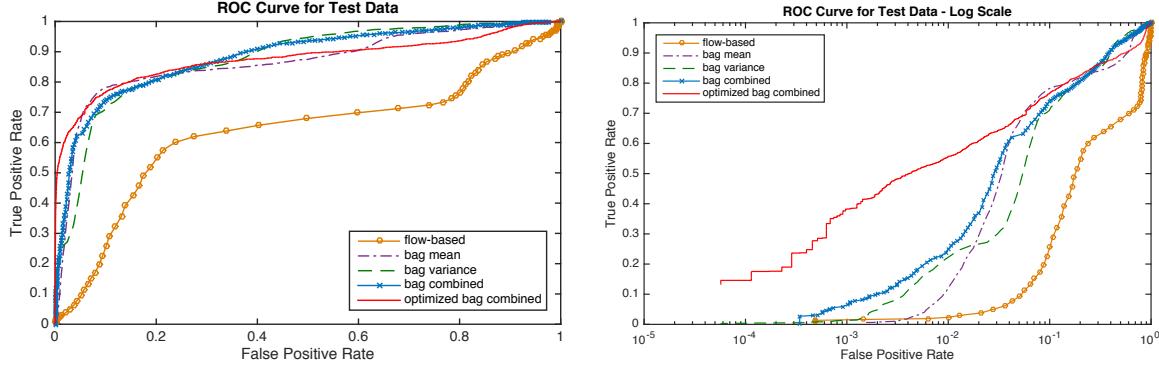


Figure 10: ROC curves of SVM classifier on test data for five types of representations (logarithmic scale on the right). Flow-based representation shows very unsatisfactory results showing that flow-based approach cannot be applied in practice to detect unseen malware variants. The combination of feature values with feature differences histogram (bag combined) led to significantly better efficacy results. These results were further exceeded when the parameters of the invariant representation were learned automatically from the training data (optimized bag combined).

plexity of the classifier.

Figures 8 and 9 show the bins and weights learned from the training set of real network traffic. The blue vertical lines represent learned weights associated with 256 bins of a histogram computed on a single input feature. The red lines show new bins derived from the weights by merging those neighboring bins which have the weights with the same sign. Figure 8 shows the weights and the derived bins for a standard SVM which has no incentive to have similar weights. The histogram derived from the SVM weights reduces the number of bins from 256 to 130. Figure 9 shows the results for the proposed method which enforces the similar weights for neighboring bins. In this case, the weights exhibit a clear structure and the derived histogram has only 18 bins. The decision boundary is in this case smoother and the classifier trained from this representation will be more robust.

Next, a two-class SVM classifier was evaluated on five representations: baseline flow-based, per-feature histograms of values $\phi(z_k^X, \theta_k^X)$ (bag mean), per-feature histograms of feature differences $\phi(z_k^S, \theta_k^S)$ (bag variance), the combination of both (bag combined), and the combination of both with bin optimization (optimized bag combined). The training and testing datasets were composed of bags described in Table 2.

The results on testing data are depicted in Figure 10. Note that positive bags in the testing set are from different malware categories than bags from the training set, which makes the classification problem much harder. The purpose of this evaluation is to compare flow-based representation, which is used in most of previously published work, with the proposed invariant representation. Flow-based representation shows very unsatisfactory results, mainly due to the fact that the classifier was based only on the values of flow-based features that are not

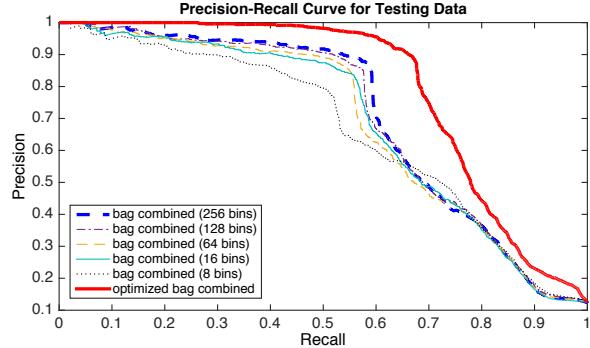


Figure 11: Precision-recall curve of SVM classifier trained on the proposed representation with different number of histogram bins for each feature. All classifiers are outperformed by the classifier, where the parameters of the invariant representation are learned automatically from the data (optimized bag combined). The classifier achieved 90% precision (9 of 10 alerts were malicious) and 67% recall on previously unseen malware families.

robust across different malware categories (as shown in Section 7). The classifier based on combined bag representation performed significantly better. These results were further exceeded when the parameters of the invariant representation were learned automatically from the training data (using bin optimization from Section 5), which is shown in Figure 10 with logarithmic scale.

Precision-recall curve is depicted in Figure 11 to compare the efficacy results of classifiers based on the proposed representation with predefined number of bins per feature (8, 16, 64, 128, and 256 bins) with the same representation, but when the parameters are learned from the training data (using bin optimization from Section 5).

Overall, the results show the importance of combining both types of histograms introduced in Section 4 together, allowing the representation to be more descriptive and precise without sacrificing recall. But most importantly, when the parameters of the representation are trained to maximize the separability between malicious and legitimate samples, the resulting classifier performs in order of a magnitude better than a classifier with manually predefined parameters.

9 Conclusion

This paper proposes a robust representation suitable for classifying evolving malware behaviors. It groups sets of network flows into bags and represents them using a combination of invariant histograms of feature values and feature differences. The representation is designed to be invariant under shifting and scaling of the feature values and under permutation and size changes of the bags. The proposed optimization method learns the parameters of the representation automatically from the training data, allowing the classifiers to create robust models of malicious behaviors capable of detecting previously unseen malware variants and behavior changes.

The proposed representation was deployed on corporate networks and evaluated on real HTTP network traffic with more than 43k malicious samples and more than 15M samples overall. The comparison with a baseline flow-based approach and a widely-used signature-based web security device showed several key advantages of the proposed representation. First, the invariant properties of the representation result in the detection of new types of malware. More specifically, the proposed classifier trained on the optimized representation achieved 90% precision (9 of 10 alerts were malicious) and detected 67% of malware samples of previously unseen types and variants. Second, multiple malware behaviors can be represented in the same feature space while current flow-based approaches necessitate training a separate detector for each malware family. This way, the proposed system considerably increases the capability of detecting new variants of threats.

References

- [1] Cisco netflow. <http://www.cisco.com/warp/public/732/tech/netflow>.
- [2] ANTONAKAKIS, M., PERDISCI, R., NADJI, Y., VASILOGLOU, N., ABU-NIMEH, S., LEE, W., AND DAGON, D. From throw-away traffic to bots: Detecting the rise of dga-based malware. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 24–24.
- [3] BAILEY, M., OBERHEIDE, J., ANDERSEN, J., MAO, Z., JAHA-NIAN, F., AND NAZARIO, J. Automated classification and analysis of internet malware. In *Recent Advances in Intrusion Detection*, C. Kruegel, R. Lippmann, and A. Clark, Eds., vol. 4637 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 178–197.
- [4] BEN-DAVID, S., BLITZER, J., CRAMMER, K., PEREIRA, F., ET AL. Analysis of representations for domain adaptation. *Advances in neural information processing systems 19* (2007), 137.
- [5] BERNAILLE, L., TEIXEIRA, R., AKODKENOU, I., SOULE, A., AND SALAMATIAN, K. Traffic classification on the fly. *ACM SIGCOMM '06* 36, 2 (Apr. 2006), 23–26.
- [6] BILGE, L., BALZAROTTI, D., ROBERTSON, W., KIRDA, E., AND KRUEGEL, C. Disclosure: Detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 129–138.
- [7] BLITZER, J., McDONALD, R., AND PEREIRA, F. Domain adaptation with structural correspondence learning. In *Proceedings of the 2006 conference on empirical methods in natural language processing* (2006), Association for Computational Linguistics, pp. 120–128.
- [8] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *ACM Comput. Surv.* 41 (July 2009), 15:1–15:58.
- [9] CHOI, H., ZHU, B. B., AND LEE, H. Detecting malicious web links and identifying their attack types. In *Proceedings of the 2Nd USENIX Conference on Web Application Development* (Berkeley, CA, USA, 2011), WebApps'11, USENIX Association, pp. 11–11.
- [10] DAI, W., YANG, Q., XUE, G.-R., AND YU, Y. Boosting for transfer learning. In *Proceedings of the 24th international conference on Machine learning* (2007), ACM, pp. 193–200.
- [11] DUAN, L., TSANG, I. W., AND XU, D. Domain transfer multiple kernel learning. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 34, 3 (2012), 465–479.
- [12] ERMAN, J., ARLITT, M., AND MAHANTI, A. Traffic classification using clustering algorithms. In *Proceedings of the 2006 SIGCOMM Workshop on Mining Network Data* (New York, NY, USA, 2006), MineNet '06, ACM, pp. 281–286.
- [13] FALLIERE, N. Sality: Story of a peer-to-peer viral network. *Report technique, Symantec Corporation* (2011).
- [14] GRETTON, A., SMOLA, A., HUANG, J., SCHMITTFULL, M., BORGWARDT, K., AND SCHÖLKOPF, B. Covariate shift by kernel mean matching. *Dataset shift in machine learning* 3, 4 (2009), 5.
- [15] GRIFFIN, K., SCHNEIDER, S., HU, X., AND CHIEH, T.-C. Automatic generation of string signatures for malware detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2009), RAID '09, Springer-Verlag, pp. 101–120.
- [16] GU, G., PERDISCI, R., ZHANG, J., LEE, W., ET AL. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In *USENIX Security Symposium* (2008), vol. 5, pp. 139–154.
- [17] HUANG, H., QIAN, L., AND WANG, Y. A svm-based technique to detect phishing urls. *Information Technology Journal* 11, 7 (2012), 921–925.
- [18] INVERNIZZI, L., MISKOVIC, S., TORRES, R., SAHA, S., LEE, S., MELLIA, M., KRUEGEL, C., AND VIGNA, G. Nazca: Detecting malware distribution in large-scale networks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2014).

- [19] IYER, A., NATH, S., AND SARAWAGI, S. Maximum mean discrepancy for class ratio estimation: Convergence bounds and kernel selection. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)* (2014), pp. 530–538.
- [20] JAGPAL, N., DINGLE, E., GRAVEL, J.-P., MAVROMMATHIS, P., PROVOS, N., RAJAB, M. A., AND THOMAS, K. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 579–593.
- [21] JUNEJO, I. N., DEXTER, E., LAPTEV, I., AND PEREZ, P. View-independent action recognition from temporal self-similarities. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 33, 1 (2011), 172–185.
- [22] KAPRAVELOS, A., SHOSHITAISHVILI, Y., COVA, M., KRUEGEL, C., AND VIGNA, G. Revolver: An automated approach to the detection of evasive web-based malware. In *USENIX Security* (2013), Citeseer, pp. 637–652.
- [23] KARAGIANNIS, T., PAPAGIANNAKI, K., AND FALOUTSOS, M. Blinc: Multilevel traffic classification in the dark. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2005), SIGCOMM ’05, ACM, pp. 229–240.
- [24] KIM, H., CLAFFY, K., FOMENKOV, M., BARMAN, D., FALOUTSOS, M., AND LEE, K. Internet traffic classification demystified: Myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT Conference* (New York, NY, USA, 2008), CoNEXT ’08, ACM, pp. 11:1–11:12.
- [25] KRUEGEL, C., AND VIGNA, G. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS ’03, ACM, pp. 251–261.
- [26] LOU, W., LIU, G., LU, H., AND YANG, Q. Cut-and-pick transactions for proxy log mining. In *Advances in Database Technology EDBT 2002*, C. Jensen, S. altenis, K. Jeffery, J. Pokorny, E. Bertino, K. Bhn, and M. Jarke, Eds., vol. 2287 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002, pp. 88–105.
- [27] MA, J., SAUL, L. K., SAVAGE, S., AND VOELKER, G. M. Learning to detect malicious urls. *ACM Trans. Intell. Syst. Technol.* 2, 3 (May 2011), 30:1–30:24.
- [28] MOORE, D., SHANNON, C., BROWN, D. J., VOELKER, G. M., AND SAVAGE, S. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.* 24, 2 (May 2006), 115–139.
- [29] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP ’07. IEEE Symposium on* (May 2007), pp. 231–245.
- [30] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (Dec 2007), pp. 421–430.
- [31] MÜLLER, M., AND CLAUSEN, M. Transposition-invariant self-similarity matrices. In *In Proceedings of the 8th International Conference on Music Information Retrieval (ISMIR)* (2007), pp. 47–50.
- [32] NELMS, T., PERDISCI, R., ANTONAKAKIS, M., AND AHAMAD, M. Webwitness: Investigating, categorizing, and mitigating malware download paths. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 1025–1040.
- [33] PORTNOY, L., ESKIN, E., AND STOLFO, S. Intrusion detection with unlabeled data using clustering. In *In Proceedings of ACM CSS Workshop on Data Mining Applied to Security (DMSA-2001)* (2001), pp. 5–8.
- [34] RIECK, K., HOLZ, T., WILLEMS, C., DSSEL, P., AND LASKOV, P. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, D. Zamboni, Ed., vol. 5137 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 108–125.
- [35] RNDIC, N., AND LASKOV, P. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on* (May 2014), pp. 197–211.
- [36] SCARFONE, K., AND MELL, P. Guide to intrusion detection and prevention systems (ids) recommendations of the national institute of standards and technology. *Nist Special Publication 800*, 94 (2007).
- [37] SHIMODAIRA, H. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference* 90, 2 (2000), 227–244.
- [38] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. Bitblaze: A new approach to computer security via binary analysis. In *Information Systems Security*, R. Sekar and A. Pujari, Eds., vol. 5352 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 1–25.
- [39] SONG, H., AND TURNER, J. Toward advocacy-free evaluation of packet classification algorithms. *Computers, IEEE Transactions on* 60, 5 (May 2011), 723–733.
- [40] SOSKA, K., AND CHRISTIN, N. Automatically detecting vulnerable websites before they turn malicious. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 625–640.
- [41] WANG, K., AND STOLFO, S. Anomalous payload-based network intrusion detection. In *Recent Advances in Intrusion Detection*, E. Jonsson, A. Valdes, and M. Almgren, Eds., vol. 3224 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 203–222.
- [42] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS ’07, ACM, pp. 116–127.
- [43] ZHANG, K., SCHÖLKOPF, B., MUANDET, K., AND WANG, Z. Domain adaptation under target and conditional shift. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* (2013), S. Dasgupta and D. Mcallester, Eds., vol. 28, JMLR Workshop and Conference Proceedings, pp. 819–827.
- [44] ZHAO, P., AND HOI, S. C. Cost-sensitive online active learning with application to malicious url detection. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2013), KDD ’13, ACM, pp. 919–927.

A Examples of Bags

Asterope	<pre>hxxp://194.165.16.146:8080/pgt/?ver=1.3.3398&id=126&r=12739868&os=6.1—2—8.0.7601.18571&res=4—1921—466&f=1 hxxp://194.165.16.146:8080/pgt/?ver=1.3.3398&id=126&r=15425581&os=6.1—2—8.0.7601.18571&res=4—1921—516&f=1 hxxp://194.165.16.146:8080/pgt/?ver=1.3.3398&id=126&r=27423103&os=6.1—2—8.0.7601.18571&res=4—1921—342&f=1 hxxp://194.165.16.146:8080/pgt/?ver=1.3.3753&id=126&r=8955018&os=6.1—2—8.0.7601.18571&res=4—1921—319&f=1</pre>
Click-fraud, malvertising-related botnet	<pre>hxxp://directcashfunds.com/opntrk.php?key=024f9730e23f8553c3e5342568a70300&Email=name.surname@company.com hxxp://directcashfunds.com/opntrk.php?key=c1b6e3d50632d4f5c0ae13a52d3c4d8d&Email=name.surname@company.com hxxp://directcashfunds.com/opntrk.php?key=7c9a843ce18126900c46dbe4be3b6425&Email=name.surname@company.com hxxp://directcashfunds.com/opntrk.php?key=c1b6e3d50632d4f5c0ae13a52d3c4d8d&Email=name.surname@company.com</pre>
DGA	<pre>hxxp://uvyqifymelapuwoh.biz/s31ka.ji5 hxxp://uvyqifymelapuwoh.biz/r159c281.x19 hxxp://uvyqifymelapuwoh.biz/seibpn6.2m0 hxxp://uvyqifymelapuwoh.biz/3854f.u17</pre>
Dridex	<pre>hxxp://27.54.174.181/8qV578&\$o@HU6Q6S/gz\$J0l=iTTH 28%2CM/we20%3D hxxp://27.54.174.181/C4GyRx%7E@RY6x /M&N=sq/bW_ra4OTJ hxxp://27.54.174.181/gPvh+=GO/9RPPfk0%2CzXOYU%20/Vq8Ww/+a.m%7Ez hxxp://27.54.174.181/qE0my4K1z48Cf3H8wG%7Evpz=iJ%26fqMI%24m/46JoELp=GJww%3D%26lb+Ar.y3 iu%2D1E/sso</pre>
InstallCore	Monetization
<pre>hxxp://rp.any-file-opener.org/?pcrc=1559319553&v=2.0 hxxp://rp.any-file-opener.org/?pcrc=1132521307&v=2.0 hxxp://rp.any-file-opener.org/?pcrc=1123945956&v=2.0 hxxp://rp.any-file-opener.org/?pcrc=1075608192&v=2.0</pre>	<pre>hxxp://utouring.net/search/q/conducting hxxp://utouring.net/go/u/l/r/1647 hxxp://utouring.net/go/u/l/r/2675 hxxp://utouring.net/search/f/1/q/refiles</pre>
Poweliks	
<pre>hxxp://31.184.194.39/query?version=1.7&sid=793&builddate=114&q=nitric+oxide+side+effects&ua=Mozilla%2F5 ... &lr=7&ls=0 hxxp://31.184.194.39/query?version=1.7&sid=793&builddate=114&q=weight+loss+success+stories&ua=Mozilla%2F5 ... &lr=0&ls=0 hxxp://31.184.194.39/query?version=1.7&sid=793&builddate=114&q=shoulder+pain&ua=Mozilla%2F5 ... &lr=7&ls=2 hxxp://31.184.194.39/query?version=1.7&sid=793&builddate=114&q=cheap+car+insurance&ua=Mozilla%2F5 ... &lr=7&ls=2</pre>	
Zeus	
<pre>hxxp://130.185.106.28/m/lbQFdXVjiriLva4KHeNpWCmThrJBn3f34HNwsLVVsUmLxtsumSSPe/zzXtlu9Szwl9zKlxde ... 3RqvGzKN5 hxxp://130.185.106.28/m/lbQJFUVjgZn4vx4KHeNpWCmThrJBn3f34HNwsLVVsUmLfkoPaSS+S+zzXtlu9Szwl9zKlxde ... 3vKwmk0oUi hxxp://130.185.106.28/m/lbQJFUVjJwJBX4KHeNpWCmThrJBn3f34HNwsLVVsUmKH7ue2STvSkzzXtlu9Szwl9zKlxde ... 3vKwmk0oUi hxxp://130.185.106.28/m/lbQNtVVji5/7Yp4KHeNpWCmThrJBn3f34HNwsLVVsUmLz4sO6YRvOjzzXtlu9Szwl9zKlxde ... 3zB9057quqv</pre>	
Legitimate traffic 1	
<pre>hxxp://www.cnn.com/.element/ssi/auto/4.0/sect/MAIN/markets_wsod_expansion.html hxxp://www.cnn.com/a/1.73.0/assets/sprite-s1dcde3ff2b.png hxxp://www.cnn.com/.element/widget/video/videoapi/api/latest/js/CNNVideoBootstrapper.js hxxp://www.cnn.com/jsonp/video/nowPlayingSchedule.json?callback=nowPlayingScheduleCallbackWrapper&_=1422885578476</pre>	
Legitimate traffic 2	
<pre>hxxp://ads.adaptv.advertising.com/a/h/7g_.doK40WLPMYHbkD9G2u7HSXjqzIaa7Bqhslod+u7iQl ... &context=fullUrl%3Dpandora.com hxxp://ads.adaptv.advertising.com/crossdomain.xml hxxp://ads.advertising.com/411f1e96-3bde-4d85-b17e-63749e5f0695.js hxxp://ads.adaptv.advertising.com/applist?placementId=297920&key=&d.vw=1&orgId=8656&hostname=data.rtbfy.com</pre>	

Table 4: Example URLs of flows from several malicious bags and from two legitimate bags. The URLs within each malicious bag are similar to each other while the URLs within legitimate bags differ. The small non-zero variability of flow-based feature values is captured by the proposed representation using histograms of features and feature self-similarity matrices. Such transformation of the feature values makes the representation robust to malware changes and unseen variants.

Authenticated Network Time Synchronization

Benjamin Dowling

Queensland University of Technology

b1.dowling@qut.edu.au

Douglas Stebila

McMaster University

stebilad@mcmaster.ca

Greg Zaverucha

Microsoft Research

gregz@microsoft.com

Abstract

The Network Time Protocol (NTP) is used by many network-connected devices to synchronize device time with remote servers. Many security features depend on the device knowing the current time, for example in deciding whether a certificate is still valid. Currently, most services implement NTP without authentication, and the authentication mechanisms available in the standard have not been formally analyzed, require a pre-shared key, or are known to have cryptographic weaknesses. In this paper we present an authenticated version of NTP, called ANTP, to protect against desynchronization attacks. To make ANTP suitable for large-scale deployments, it is designed to minimize server-side public key operations by infrequently performing a key exchange using public key cryptography, then relying solely on symmetric cryptography for subsequent time synchronization requests; moreover, it does so without requiring server-side per-connection state. Additionally, ANTP ensures that authentication does not degrade accuracy of time synchronization. We measured the performance of ANTP by implementing it in OpenNTPD using OpenSSL. Compared to plain NTP, ANTP’s symmetric crypto reduces the server throughput (connections/second) for time synchronization requests by a factor of only 1.6. We analyzed the security of ANTP using a novel provable security framework that involves adversary control of time, and show that ANTP achieves secure time synchronization under standard cryptographic assumptions; our framework may also be used to analyze other candidates for securing NTP.

Keywords: time synchronization, Network Time Protocol (NTP), provable security, network security

1 Introduction

The *Network Time Protocol (NTP)* is one of the Internet’s oldest protocols, dating back to RFC 958 [15] published in 1985. In the simplest NTP deployment, a client device

sends a single UDP packet to a server (the *request*), who responds with a single packet containing the time (the *response*). The response contains the time the request was received by the server, as well as the time the response was sent, allowing the client to estimate the network delay and set their clock. If the network delay is *symmetric*, i.e., the travel time of the request and response are equal, then the protocol is perfectly accurate. *Accuracy* means that the client correctly synchronizes its clock with the server (regardless of whether the server clock is accurate in the traditional sense, e.g., synchronized with UTC).

The importance of accurate time for security. There are many examples of security mechanisms which (often implicitly) rely on having an accurate clock:

- *Certificate validation in TLS and other protocols.* Validating a public key certificate requires confirming that the current time is within the certificate’s validity period. Performing validation with a slow or inaccurate clock may cause expired certificates to be accepted as valid. A revoked certificate may also validate if the clock is slow, since the relying party will not check for updated revocation information.
- *Ticket verification in Kerberos.* In Kerberos, authentication tickets have a validity period, and proper verification requires an accurate clock to prevent authentication with an expired ticket.
- *HTTP Strict Transport Security (HSTS) policy duration.* HSTS [10] allows website administrators to protect against downgrade attacks from HTTPS to HTTP by sending a header to browsers indicating that HTTPS must be used instead of HTTP. HSTS policies specify the duration of time that HTTPS must be used. If the browser’s clock jumps ahead, the policy may expire re-allowing downgrade attacks. A related mechanism, *HTTP Public Key Pinning* [7] also relies on accurate client time for security.

For clients who set their clocks using NTP, these security mechanisms (and others) can be attacked by a

network-level attacker who can intercept and modify NTP traffic, such as a malicious wireless access point or an insider at an ISP. In practice, most NTP servers do not authenticate themselves to clients, so a network attacker can intercept responses and set the timestamps arbitrarily. Even if the client sends requests to multiple servers, these may all be intercepted by an upstream network device and modified to present a consistently incorrect time to a victim. Such an attack on HSTS was demonstrated by Selvi [28], who provided a tool to advance the clock of victims in order to expire HSTS policies. Malhotra et al. [12] present a variety of attacks that rely on NTP being unauthenticated, further emphasizing the need for authenticated time synchronization. (Confidentiality, however, is not a requirement for time synchronization, since all time synchronization is public. Similarly, client-to-server authentication is not a goal.)

NTP security today. Early versions of NTP had no standardized authentication method. NTPv3 added an authentication method using pre-shared key symmetric cryptography. An extension field in the NTP packet added a cryptographic checksum, computed over the packet. In NTPv3 negotiation of keys and algorithms must be done out-of-band. For example, NIST offers a secure time server, and (symmetric) keys are transported from server to client by postal mail [21]. Establishing pre-shared symmetric keys with billions of client PCs and other NTP-synchronizing devices would be impractical. NTPv4 introduced a public key authentication mechanism called Autokey which has not seen widespread adoption; and unfortunately, Autokey uses small 32-bit seeds that can be easily brute forced to then forge packets. A more recent proposal is the Network Time Security (NTS) protocol [31], which we discuss in §2.3.

Most NTP servers do not support NTP authentication, and NTP clients in desktop and laptop operating systems will set their clocks based on unauthenticated NTP responses. On Linux and OS X, by default the client either polls a server periodically, or creates an NTP request when the network interface is established. In both cases the system clock will be set to any time specified by the NTP response. On Windows, by default clients will synchronize their clock every nine hours (using `time.microsoft.com`), and ignore responses that would change the clock by more than 15 hours. These two defaults reduce the opportunity for a man-in-the-middle (MITM) attacker to change a victim clock and the amount by which it may be changed, but cumulative small-scale changes can build over time to large-scale time inaccuracies. Teichel et al. used this technique when attacking time-synchronization secured by TESLA-like protocols [33]. In Windows domains (a network of computers, often in an enterprise), the domain controller provides the time with an authenticated variant of NTPv3 [14].

1.1 Contributions

We present the ANTP protocol for authenticated network time synchronization, along with results on its performance and security. ANTP protocol messages are transported in the extension fields of NTP messages. ANTP allows a server to authenticate itself to a client using public key certificates and public key exchange, and provides cryptographic assurance using symmetric cryptography that no modification of the packets has occurred in transit. Like other authenticated time synchronization protocols using public keys [31], we assume an out-of-band method for certificate validation exists, as certificate validation requires an accurate clock. We follow the direction set by the IETF Informational document “Security Requirements of Time Protocols in Packet-Switched Networks” (RFC 7384) [20] to determine what cryptographic, computational, and storage properties ANTP should achieve.

ANTP has three phases. In the *negotiation phase*, the client and server agree on which cryptographic algorithms to use; this phase would be carried out quite infrequently, on the order of monthly or less. In the *key exchange phase*, the client and server use public key cryptography to establish a symmetric key that the server will use to authenticate later time synchronization responses; this phase would also be carried out infrequently, say monthly. In the *time synchronization phase*, the client sends a time synchronization request, and the server replies with an NTP response that is symmetrically authenticated using the key established in the key exchange phase; this may be done frequently, perhaps daily or more often. Notably, the server need not keep per-client state: the server offloads any such state to the client by encrypting and authenticating it under a long-term symmetric key, and the client sends that ciphertext back to the server with each subsequent request.

The time synchronization phase of ANTP can be run in a “no-cryptographic-latency” mode: here, the server sends two response packets, the first being the unauthenticated NTP packet, and the second being the same NTP packet (with unchanged timestamps) along with the ANTP extensions providing authentication. The client measures the roundtrip time based on the unauthenticated response, but does not update its clock until authenticating the response. In this way, no time synchronization inaccuracy is added by the time required to compute the authentication tag over the outgoing timestamp. Since the latency of ANTP’s time synchronization phase is nearly as fast as unauthenticated simple NTP time synchronization (only 21 microseconds slower at 50% load in our implementation as reported below), we make this mode optional since plain ANTP may be sufficiently accurate for general use.

ANTP performance. Performance constraints on time synchronization protocols are driven by the fact that time

Phase	Throughput	Latency within LAN (μs)		Latency across US (ms)	
		50% load	90% load	50% load	90% load
ANTP – Negotiation – RSA	58 240	186 \pm 26	202 \pm 44	76.3 \pm 0.1	77.5 \pm 0.1
ANTP – Negotiation – ECDH	146 808	172 \pm 35	233 \pm 133	75.3 \pm 0.1	75.3 \pm 0.1
ANTP – Key Exchange – RSA	1 754	891 \pm 125	997 \pm 348	75.8 \pm 0.2	76.9 \pm 0.5
ANTP – Key Exchange – ECDH	13 210	197 \pm 56	344 \pm 142	74.7 \pm 0.2	75.4 \pm 0.4
ANTP – Time Synchronization	175 644	168 \pm 35	230 \pm 160	73.5 \pm 0.1	73.7 \pm 0.1
ANTP – All 3 phases – RSA	–	2255 \pm 587	2646 \pm 345	226.6 \pm 6.2	258.0 \pm 35
ANTP – All 3 phases – ECDH	–	1325 \pm 499	2252 \pm 1 172	231.8 \pm 10.5	223.3 \pm 6.7
NTP	291 926	147 \pm 34	181 \pm 136	72.4 \pm 0.1	74.0 \pm 0.1

Table 1: Performance results for each phase of ANTP (top), a complete 3-phase execution of ANTP (middle), and NTP (bottom). Throughput: mean completed phases per second. Latency: mean and standard deviation of the latency of server responses at either 50% or 90% server load on a local area network (reported in microseconds) and across the United States (between Virginia and California) (reported in milliseconds). All are computed over 5 trials, top and bottom over 100 seconds each; see Section 4.2 for details.

servers are heavily loaded, and must provide responses promptly. ANTP’s design allows it to achieve high performance while maintaining high security. The frequently performed time synchronization phase uses only symmetric cryptography, making it only slightly more expensive than simple NTP time synchronization. Since the session key established in the key exchange phases is reused across many time synchronization phases, expensive public key operations are amortized, and can be separately load-balanced. And, as noted above, ANTP offloads state to clients, leaving the server stateless.

We implemented ANTP in OpenNTPD’s [34] implementation of NTP, using OpenSSL’s libcrypto library (but not SSL/TLS) [35] for cryptographic computations. Table 1 reports the performance of our implementation, compared with unauthenticated simple NTP. ANTP does decrease throughput and increase latency, but the impact is quite reasonable. On a single core of a server, ANTP can support 175k authenticated time synchronization phase connections per second, a factor of 1.6 fewer than the 291k unauthenticated simple NTP connections per second. Latency for time synchronization (over a 1 gigabit per second local area network) at 50% load increases from 147 microseconds for unauthenticated simple NTP to 168 microseconds for ANTP’s time synchronization phase. The other two phases, negotiation and key exchange, will be performed far less frequently on average by clients. Throughput of negotiation phases is bandwidth-, not CPU-, limited. For exchange, we implemented methods: 2048-bit RSA key transport and static-ephemeral elliptic curve Diffie–Hellman key exchange using the NIST P-256 curve; as expected, both of these are substantially more expensive than time synchronization phases, but are also performed far less frequently.

Protocol	Auth. type	Security	Server operations	Round trips
NTPv0-v2	—	—	—	1
NTPv3	sym. key	no proof	1 hash	1
sym. key				
NTPv4	pub. key	flaws (§. 2.3)	$\frac{2}{n}$ pub. key, $\frac{1}{n} + 1$ sym. key	4
Autokey				
tsdate [3]	pub. key	relies on TLS	$\frac{2}{n}$ pub. key [†] $\frac{5}{n} + 7$ sym. key [†]	4 stateful
NTS [31]	pub. key	ProVerif proof [32]	$\frac{3}{n}$ pub. key, $\frac{2}{n} + 2$ sym. key	4
ANTP	pub. key (Fig. 3)	proof (Sec. 6)	$\frac{1}{n}$ pub. key, $\frac{6}{n} + 2$ sym. key	3

Table 2: Comparison of time synchronization protocols. Server operations per time synchronization includes public key decryptions, symmetric key encryptions/decryptions, and hashes/KDFs/MACs. $\frac{a}{n} + b$ denotes a operations that can be amortized over n time synchronizations plus b operations per time sync.

[†] tsdate operation counts vary based on ciphersuite.

Details of our implementation and testing methodology, as well as more results, appear in Section 4.

ANTP compares well with other authentication methods for NTP, as seen in Table 2. ANTP uses fewer amortized public key operations compared to NTPv4 Autokey and NTS and has fewer rounds. NTPv3 using symmetric key operations is more lightweight, but is highly restricted in that it only supports symmetric authentication via pre-established symmetric keys, making it unsuitable for deployment with billions of devices.

Because ANTP is designed-for-purpose, it is also more

efficient than applying general purpose security protocols to NTP. For example, one might consider simply applying TLS or DTLS to NTP packets to obtain authentication. Unfortunately, TLS and DTLS do not achieve full statelessness as is desirable for high throughput applications that need to resist denial of service attacks. TLS and DTLS both require that the server maintain state during the initial handshake (which requires 3 round trips) and during session resumption handshakes (which requires 2 round trips). While the server can offload state *between* the initial handshake and the session resumption using session tickets, there is no standardized mechanism to do so *during* the handshakes. DTLS servers also must maintain mappings between clients and sessions, as UDP does not provide this functionality. ANTP avoids these problems by having the server offload state at every step. ANTP is also much more efficient in terms of communication size, as TLS/DTLS have a variety of extensions that consume substantial bandwidth.

ANTP security. ANTP’s design is supported by a thorough analysis of its cryptographic security using the provable security paradigm. To do so, we extend existing frameworks for key exchange and secure channels [5, 11] to develop a novel framework that handles protocols where *time* plays a central role. The adversary in our security analysis is a network attacker capable of deleting, reordering, editing, and creating messages between parties. Since our model is about time synchronization, parties in our model have local clocks, and the adversary is given complete control over the initialization of all clocks, as well as the ability to increment the time of parties not involved in a protocol run. This allows us to model the ability of an adversary to delay packet transmission: this is particularly important in the case of NTP, where delaying packets asymmetrically can cause the client to synchronize to an inaccurate time. This differs from previous security frameworks that model time, such as ones introduced by Schwenk [27] (which uses a global time counter to model timestamps) and Basin et al. [4] (where the adversary cannot influence the offset or rate-of-change of the party clocks.)

We then show that ANTP achieves secure time synchronization as defined by our model, under standard assumptions on the security of the cryptographic primitives (key encapsulation mechanism, hash function, authenticated encryption, message authentication code, and key derivation function) used to construct the protocol.

2 Network Time Protocols

Here we review the two most commonly deployed time synchronization protocols, NTP and SNTP, as well as a recent proposal called Network Time Security [31].

2.1 The Network Time Protocol

The *Network Time Protocol (NTP)* was developed by Mills in 1985 [15], and revised in 1988, 1989, 1992 and 2010 (NTPv1 [9], NTPv2 [16], NTPv3 [17] and NTPv4 [18] respectively). NTP is designed to synchronize the clocks of machines directly connected to hardware clocks (known as *primary servers*) to machines without hardware clocks (known as *secondary servers*). NTP protects against Byzantine traitors by querying multiple servers, selecting a majority clique and updating the local clock with the majority offset. This assumes the attacker can only influence some minority of the queried servers.

2.2 The Simple Network Time Protocol

The *Simple Network time Protocol (SNTP)* is a variant of NTP that uses an identical message format [17] but only queries a single server when requesting time synchronization using a single time source (`time.windows.com` and `time.apple.com` respectively). Our construction lends itself well to SNTP, as it authenticates time samples from a single server. Security analysis is also easier as we can avoid the more complex sorting and filtering algorithms of NTP, and client and server behaviours are simpler. Note that SNTP and NTP client request messages are the same.

SNTP has three distinct stages: (1) the creation and transmission of `req` by the client; (2) the processing of `req` by the server, and transmission of `resp`; and (3) the processing of `resp` and clock update by the client. An abstraction of the protocol behaviour can be found in Figure 1, including the client’s clock update procedure. Though the format of NTP packets is identical for both client and server NTP messages, we use `req` to indicate a NTP packet in client mode, and `resp` to indicate a NTP packet in server mode, omitting packet content details.

1. The client creates an SNTP `req` packet, sets `transmit_timestamp` (t_1) to `Now()` and sends the message.¹
2. The server creates an SNTP `resp` packet with all fields identical to the received `req`, but signalling Server mode. The server sets `originate.timestamp` to the value `transmit_timestamp` from `req`. The server also sets `receive_timestamp` (t_2) to `Now()` immediately after receipt of `req`, and sets `transmit_timestamp` (t_3) to `Now()` immediately before sending the message to the client.
3. Upon receiving `resp`, the client notes the current time from `Now()` and saves it as t_4 . If `resp.originate_timestamp` is not equal to

¹`Now()` denotes a party reading its local clock’s current time.

Client	Server
$t_1 \leftarrow \text{Now}()$	
$\text{req} \leftarrow t_1$	$\xrightarrow{\text{req}}$
	$t_2 \leftarrow \text{Now}()$
	⋮
	$\xleftarrow{\text{resp}}$
$t_4 \leftarrow \text{Now}()$	$t_3 \leftarrow \text{Now}()$
$RTT \leftarrow (t_4 - t_1) - (t_3 - t_2)$	$\text{resp} \leftarrow t_2 \ t_3$
$\tilde{\theta}_3 \leftarrow RTT/2$	
$offset \leftarrow \frac{1}{2}(t_2 + t_3 - t_1 - t_4)$	
$time \leftarrow \text{Now}() + offset$	

Figure 1: Simple Network Time Protocol (SNTP). `Now()` denotes the procedure that outputs the local machine’s current time. RTT denotes the total round-trip delay the client observes and $\tilde{\theta}_3$ denotes the approximation of the propagation time from server to client. The time of the server receiving `req` is denoted t_2 and sending `resp` is t_3 . Note that $offset = t_3 + \tilde{\theta}_3 - t_4$, which we will use in our correctness analysis of ANTP.

`req.transmit_timestamp`, the client aborts the protocol run. The client calculates the total round-trip time RTT and the local clock offset $offset$ as in Figure 1.

(The rest of the fields in the NTP packets are irrelevant for calculating the local clock offset and correcting the local clock for a single-source time synchronization protocol. These extra fields in the NTP packet are used primarily for ranking multiple distinct time sources.)

From this, we can compute a bound of the amount of error that is introduced to the clock update procedure via asymmetric packet delay when the packets are unmodified. Asymmetric packet delay is the scenario where the propagation time from client to server is not equal to the propagation time from server to client. Let θ_1 be the propagation time from client to server, θ_2 the server processing time and θ_3 the propagation time from server to client. θ_3 is approximated in SNTP by $\tilde{\theta}_3 = \frac{RTT}{2}$, where $RTT = (t_4 - t_1) - (t_3 - t_2) = \theta_1 + \theta_3$.

The actual offset is $offset_{actual} = t_3 + \theta_3 - t_4$. The approximated offset is computed as $offset = \frac{1}{2}(t_2 + t_3 - t_1 - t_4)$. When $\theta_1 = \theta_3$, then $offset = t_3 + \theta_3 - t_4$ and $offset = offset_{actual}$. In the worst possible case, packet delivery is instantaneous, and the entire roundtrip time is asymmetric delay. The client approximates the offset as above, and thus the error introduced this way is $\frac{1}{2}|(\theta_1 - \theta_3)| \leq RTT$.

The error that a passive adversary with the ability to delay packets can introduce does not exceed the RTT : clients can abort the protocol run when RTT grows too large, giving them some control over the worst-case error.

2.3 NTP Security and Other Related Work

In terms of security, early versions of NTP (NTP to NTPv2) had no standardized authentication method.

NTPv3 symmetric key authentication. NTPv3 presented a method for authenticating time synchronization – using pre-shared key symmetric cryptography. NTPv3’s added additional extension fields to the NTP packet, consisting of a 32-bit key identifier, and a 64-bit cryptographic checksum. The specification of NTPv3 describes the checksum as the encryption of the NTP packet with DES, but notes that other algorithms could be negotiated. The distribution of keys and negotiation of algorithms was considered outside the scope of NTP.

NTPv4 Autokey public key authentication. NTPv4 introduced a method for using public key cryptography for authentication, known as the Autokey protocol. Autokey is designed to prevent inaccurate time synchronization by authenticating the server to the client, and verifying no modification of the packet has occurred in transit. Autokey is designed to work over the top of authenticated NTPv3. Autokey uses MD5 and a variety of Schnorr-like [26] identification schemes to prevent malicious attacks, but as an analysis of Autokey by Röttger shows [23], there are multiple weaknesses inherent in the Autokey protocol, including use of small seed values (32 bits) and allowing insecure identification schemes to be negotiated. The size of the seed allows a MITM adversary with sufficient computational power to generate all possible seed values and use the cookie to authenticate adversarial-chosen NTP packets. This weakness alone allows an attacker in control of the network to break authentication of time synchronization, thus NTP with the Autokey protocol is not a secure time synchronization protocol [30]. Mills describes his experiments on demonstrating reliability and accuracy of network time synchronization using NTPv2 implementations [19], but does not offer a formal security analysis of NTP. Mills does show that honest deployment of NTP in networks can offer time synchronization accuracy to within a few tens of milliseconds after only a few synchronizations. ANTP was originally intended as a means to addressing the vulnerabilities in the Autokey protocol, but with many changes to minimize public key and symmetric key operations, message bandwidth. While inspiration for ANTP is the Autokey protocol, the design diverged significantly enough to consider it a separate protocol design.

Network Time Security draft-12. The Network Time Security protocol (NTS) [31] is an IETF Internet-Draft that uses public key infrastructure in order to secure time synchronization protocols such as NTP and the Precision Time Protocol (PTP) [1]. However, NTS is costly in terms of server-side public key operations, is a four round-trip protocol, requires clients to manage public/private

key pairs and digital certificates, and does not have an equivalent to ANTP’s no-cryptographic-latency feature.

We note NTS is a work-in-progress and future revisions may be updated to address these issues. We discovered in draft-06 a flaw in the *Association Phase* that would allow MITM adversaries to perform downgrade attacks and communicated our findings to the authors. This has since been fixed and the following is an overview of draft-12.

NTS evolved as an attempt to fix the weaknesses in Autokey and has inherited many design choices from the Autokey protocol, in particular protocol flow and key derivation strategy using secret server seeds. Similarly to the Autokey protocol, NTS servers reuse the randomness *server_seed* used to generate a shared secret key (referred to as a *cookie*) for each client by $\text{cookie} = \text{HMAC}(\text{server_seed}, \text{Hash}(\text{client public key certificate}))$, encrypting this value and a client-chosen nonce with the client public key, authenticating the server by digitally signing the *cookie* with the server private key. Note that the client public key certificate in NTS serves to ensure the confidentiality and uniqueness of the *cookie* for each client using a different public key certificate. It does not serve to authenticate the client to the server. In ANTP clients do not need a certificate, only the server.

In the *Association Phase* NTS requires the server digitally sign the server *assoc* message, which (in draft-12) includes the client’s selection of hash and public key encryption algorithms in addition to a client nonce. The server must compute costly public key operations over these values for each association phase. As a result, a NTS server requires three public key operations per client to establish a shared secret *cookie*.

NTS draft-06 attack. Here we briefly describe our downgrade attack on NTS draft-06 and below. Figure 2 shows the NTS draft-06 *Association Phase*, which differs to the previously described draft-12. (This is analogous to the *Negotiation Phase* in ANTP.) In assoc_c , the client sends the highest version of NTS that it supports, in addition to lists of hash and public key encryption algorithms. The server responds with a signature over the server hostname, the negotiated version, and the negotiated hash and public key encryption algorithms. This does not contain the server’s received values of the client’s supported version and algorithms, so it is straightforward for a MITM attacker to strip assoc_c of strong hash and public key encryption algorithms. If the client supports weak algorithms, this can translate into a full break of time-synchronization security by recovery of the *cookie*, similarly to attacks on weak Diffie-Hellman groups in TLS [2].

tlodate. Another solution for synchronizing time is Appelbaum’s *tlodate* [3], which uses timestamps in the nonces of TLS handshakes to fetch the time. However, this requires stateful servers, is not as accurate as cur-

rent solutions and will no longer function when servers transition to TLS 1.3 (as the proposed protocol no longer includes timestamps in the nonces).

3 Authenticated NTP

In this section we present the *Authenticated Network Time Protocol (ANTP)*: a new variant of NTP designed to allow an SNTP client to authenticate a single NTP server and output a time counter within some accuracy margin of the server time counter. Our new protocol ANTP allows an ANTP server to authenticate itself to an ANTP client, as well as provide cryptographic assurances that no modification of the packets has occurred in transit. ANTP messages, much like Autokey and NTS, are included in the extension fields of NTP messages. We summarize the novel features of ANTP below:

- The client is capable of authenticating the server, and all messages from the server. Replay attacks are explicitly prevented for the client.
- The server does not need to keep state for each client.
- The server does only one public key operation per client in order to generate a shared secret key.
- The shared secret key can be used for multiple time synchronization attempts by the same client.
- The client has a “no-cryptographic-latency” option to avoid additional error in the approximation of $\tilde{\theta}_3$ due to cryptographic operations.

3.1 Protocol Description

ANTP is divided into four separate phases. A detailed protocol flow can be found in Figure 3, and exact message formatting can be found in the full version [6].

- *Setup:* The server chooses a long term key s for the authenticated encryption algorithm. This is used to encrypt and authenticate offloaded server state between phases.
- *Negotiation Phase:* The client and server communicate supported algorithms; the server sends its certificate and state C_1 , an authenticated encryption (using s) of the hash of the message flow. The value C_1 will be later used to authenticate negotiation.
- *Key Exchange Phase:* The client uses a key encapsulation mechanism (KEM) based on the server’s public key from its certificate to establish a shared key with the server. The client sends the KEM ciphertext and encrypted state C_1 to the server. The server derives the shared key k , then encrypts it (using s) to compute C_2 . The server replies with a MAC (for key confirmation) and offloaded state C_2 (for use in the next phase).

Client	Server
<i>Association phase</i>	
$\text{assoc}_c \leftarrow \text{vers}_c \parallel \text{name}_c \parallel \vec{\text{Hash}}_c \parallel \vec{\text{Enc}}_c$	$\xrightarrow{\text{assoc}_c} (\text{vers}_n, \text{Hash}, \text{Enc}) \leftarrow \text{negotiate}(\text{assoc}_c, \text{vers}_s, \vec{\text{Hash}}_s, \vec{\text{Enc}}_s)$ $\sigma \leftarrow \text{Sign}(\text{sk}_s, \text{vers}_n \parallel \text{name}_s \parallel \text{Hash} \parallel \text{Enc})$ $\text{assoc}_s \leftarrow \text{vers}_n \parallel \text{name}_s \parallel \text{Hash} \parallel \text{Enc} \parallel \sigma \parallel \text{cert}_s$

Figure 2: A description of the Network Time Security draft-06 Association Phase. vers is the NTS version indicator; name_c and name_s are the hostnames of the client and server respectively.

- *Time Synchronization Phase:* The client sends a time synchronization request and includes offloaded server state C_2 . The server recovers shared key k from C_2 and uses it to authenticate the response, which the client verifies. The client can also request “no-cryptographic-latency” time synchronization, where the server will immediately reply without authentication, and then send a second message with authentication.

3.2 Design Rationale and Discussion

Of the security properties discussed in RFC 7384 [20], ANTP achieves the following: *protection against manipulation, spoofing, replay and delay attacks; authentication of the server* (if ANTP is applied in a chain, implicit authentication of primary server); *key freshness; avoids degradation of time synchronization; minimizes computational load; minimizes per-client storage requirements of the server*. The following properties from [20] are only partly addressed by ANTP, which we explain in further detail below: resistance against the *rogue master, cryptographic DoS, and time-protocol DoS* attacks.

Stateless server. While storage costs are generally not an issue, synchronizing state between multiple servers implementing a high-volume network endpoint like `time.windows.com` is still expensive and complicated to deploy. For reliability and performance these servers are often in multiple data centres, spread across multiple geographic regions. In ANTP the server regenerates per-client state as needed. Our construction uses *authenticated encryption (AE)* in a similar manner to TLS Session Tickets [24] for session resumption, where the server authenticates and encrypts its per-client state using a long-term symmetric key, then sends the ciphertext to the client for storage. The client responds with the ciphertext in order for the server to decrypt and recover state. The server periodically refreshes the long-term secret key for the AE scheme (the intervals are dependent on the security requirements of the AE scheme).

No-cryptographic-latency mode. In SNTP, the accuracy is bounded by the total roundtrip time of the time

synchronization phase. If we build a secure authentication protocol over SNTP, then the total accuracy of the new authenticated protocol is also bound by the total round-trip time of the time synchronization phase.

Since cryptographic computations over the synchronization messages adds asymmetrically to propagation time, it introduces error in the approximation of propagation time $\tilde{\theta}_3$, so authentication operations degrade the accuracy of the `transmit_timestamp` in the `resp`. As noted above, ANTP includes a “no-cryptographic-latency” mode to reduce error due to authentication: during the Time Synchronization Phase, at the client’s option, the server will immediately process a `resp` as in Figure 1 and sends it to the client, without authentication. The server subsequently creates an ANTP `ServerResp` message, and sends the `resp` with `ServerResp` in the NTP extension fields of the saved `resp`. A client can then use the time when receiving the initial `resp` to set its clock, but only after verifying authentication with the ANTP `ServerResp`, aborting if authentication fails, if either message wasn’t received, or if messages were received in incorrect order. Here, cryptographic processing time does not introduce asymmetric propagation time. (The TESLA broadcast authentication protocol of Perrig et al. [22] delays authentication as well, to improve efficiency rather than accuracy as in ANTP.)

Efficient cryptography. Public key operations are computationally expensive, especially in the case of a server servicing a large pool of NTP clients. ANTP only requires a single public key operation per-client to ensure authentication and confidentiality of the premaster secret key material. The client can reuse the shared secret key on multiple subsequent time synchronization requests with that server. ANTP uses a *key encapsulation mechanism* for establishing the shared secret key. We allow either static-ephemeral elliptic curve Diffie-Hellman key exchange or key transport using RSA public key encryption. While one might ordinarily avoid use of RSA or static-ephemeral DH for key exchange since they do not provide forward secrecy, this is not a concern for ANTP since we do not need confidentiality as the contents of the messages (time synchronization data) are public.

Client	Server
supported algorithms \vec{alg}_C	supported algorithms \vec{alg}_S long-term secret s certificate $cert_S$ for the KEM keypair (pk_S, sk_S)
<i>Negotiation phase</i>	
$\alpha \leftarrow \text{in-progress}$	
$n_c \leftarrow \{0, 1\}^{256}$	
$m_1 \leftarrow \vec{alg}_C \ n_c$	$\xrightarrow{m_1}$ $(\text{KDF}, \text{Hash}, \text{KEM}, \text{MAC}) \leftarrow \text{negotiate}(\vec{alg}_C, \vec{alg}_S)$ $h \leftarrow \text{Hash}(m_1 \ \vec{alg}_S \ cert_S)$ $C_1 \leftarrow \text{AuthEnc}_S(01 \ h \ \text{KDF} \ \text{Hash} \ \text{KEM} \ \text{MAC})$
Verify $cert_S$	$\xleftarrow{m_2} m_2 \leftarrow \vec{alg}_S \ cert_S \ C_1$
$pk_S \leftarrow \text{parse}(cert)$	
<i>Key exchange phase</i>	
$(\text{KDF}, \text{Hash}, \text{KEM}, \text{MAC}) \leftarrow \text{negotiate}(\vec{alg}_C, \vec{alg}_S)$	
$h \leftarrow \text{Hash}(m_1 \ \vec{alg}_S \ cert_S)$	
$(e, pms) \leftarrow \text{KEM.Encap}(pk_S)$	
$m_3 \leftarrow C_1 \ e$	$\xrightarrow{m_3}$ $b \ h \ \text{KDF} \ \text{Hash} \ \text{KEM} \ \text{MAC} \leftarrow \text{AuthDec}_S(C_1)$ If $b \neq 01$, then $\alpha \leftarrow \text{reject}$ and abort $pms \leftarrow \text{KEM.Decap}(sk_S, e)$ $k \leftarrow \text{KDF}(pms, \perp, \text{"ANTP"}, len)$ $C_2 \leftarrow \text{AuthEnc}_S(02 \ k \ \text{KDF} \ \text{Hash} \ \text{KEM} \ \text{MAC})$ $\tau_1 \leftarrow \text{MAC}(k, h \ m_3 \ C_2)$
Verify $\tau_1 = \text{MAC}(k, h \ m_3 \ C_2)$	$\xleftarrow{m_4} m_4 \leftarrow C_2 \ \tau_1$
If verify fails, then $\alpha \leftarrow \text{reject}$ and abort	
<i>Time synchronization phase</i> $p = 1, \dots, n$	
$\alpha \leftarrow \text{in-progress}$	
$n_{c_2} \leftarrow \{0, 1\}^{256}$	
$t_1 \leftarrow \text{Now}()$	
$m_5 \leftarrow t_1 \ n_{c_2} \ C_2$	$\xrightarrow{m_5}$ $t_2 \leftarrow \text{Now}()$ $b \ k \ \text{KDF} \ \text{Hash} \ \text{KEM} \ \text{MAC} \leftarrow \text{AuthDec}_S(s, C_2)$ If $b \neq 02$, then $\alpha \leftarrow \text{reject}$ or abort $t_3 \leftarrow \text{Now}()$ $\left[\xleftarrow{m_6^*} \right]$ $m_6^* \leftarrow t_1 \ t_2 \ t_3$ $\tau_2 \leftarrow \text{MAC}(k, m_5 \ t_1 \ t_2 \ t_3)$ $m_6 \leftarrow t_1 \ t_2 \ t_3 \ \tau_2$
$t_4 \leftarrow \text{Now}()$	
$RTT \leftarrow (t_4 - t_1) - (t_3 - t_2)$	
If $RTT > E$, then $\alpha \leftarrow \text{reject}$ and abort	
Verify $\tau_2 = \text{MAC}(k, m_5 \ t_1 \ t_2 \ t_3)$	
If verify fails, then $\alpha \leftarrow \text{reject}$ and abort	
$offset = \frac{1}{2}(t_3 + t_2 - t_1 - t_4)$	
$time_p \leftarrow \text{Now}() + offset$	
$\alpha \leftarrow \text{accept}_p$	
If $p = n$, then terminate	

Figure 3: Authenticated NTP (ANTP_E), where E is a fixed upper bound on the desired accuracy. The pre-determined negotiation function negotiate takes as input two ordered lists of algorithms and returns a single algorithm. n denotes the maximum number of synchronization phases, and p denotes the current synchronization phase. $[m_6^*]$ indicates an optional message sent based on a “no-cryptographic-latency” flag present in m_5 , omitted in this figure. Note that if KEM.Decap or AuthDec fails for any ANTP server, the server simply stops processing the message, aborts, and allows the client to time-out. If certificate validation fails, the client aborts the protocol run. Each of the messages contains an identifier flag to prevent confusion between MACs. The protocol in the figure is an abstraction: messages $m_1 \dots m_4$ and the cryptographic components of m_5 and m_6 are sent as extensions of NTP messages, and detailed message structure can be found in the full version [6]. Authenticated Encryption schemes are not negotiated as they are entirely opaque to the client.

Key freshness and reuse. ANTP allows multiple time synchronization phases for each session using the same shared secret key k but with a new nonce in each Time Synchronization Phase to prevent replay attacks and ensure uniqueness of the protocol flow. This reuse can continue until either the client restarts the negotiation phase or the server rotates public keys or authenticated encryption keys.

Denial of service attacks. Against a man-in-the-middle, some types of denial-of-service (DoS) attacks are unavoidable, as the adversary may always drop messages.

Amplification attacks can be of concern. Unauthenticated SNTP has a roughly 1:1 ratio of attacker work to server work, in that one attack packet causes one packet in response, and a small computational effort is required by the server. In ANTP, the cryptographic operations do allow some amplification of work. Based on the experimental results in Table 1, the negotiation and time synchronization phases have less than a 1:2 ratio of attacker work to server work. As for the key exchange phase, the server performs a public key operation while a malicious client may not. However, a server under attack can temporarily stop responding to key exchange requests while still responding to time synchronization requests, and since most honest clients will perform key exchange infrequently, their service will not be denied.

Another amplification can be caused by the no-cryptographic-latency feature, since two response packets are sent for each request. This mode can be turned off during attack, the server indicating with a flag that it does not (currently) support this feature.

Finally, in the negotiation phase the server’s response is also considerably larger than the client request (because it includes a certificate), but, like the key exchange phase, the negotiation phase may be temporarily disabled without denying service to clients who already have established a premaster secret. Another option is to replace the server certificate chain with a URL where the client can download it. Depending on the size of the certificate(s) this could reduce the bandwidth amplification considerably. This last mitigation requires detailed analysis, which we leave to future work.

Certificate validation. When using digital certificates to authenticate public keys, the synchronization of the issuer and the relying party is an underlying assumption. This serves to highlight a significant problem – *how do you securely authenticate time using public key infrastructure without previously having time synchronization with the issuer?* For our construction this must be done once, and we assume that the client has some out-of-band method for establishing the trustworthiness of public keys, perhaps using OCSP [25] with nonces to ensure freshness of responses, by the user manually setting the time for first certificate validation, or shipping a trusted certificate with

the operating system. Since certificate validity periods typically range from months to years, if the user is assured of time synchronization with the issuer to be within range of hours or days and that range sits comfortably within the certificate validation period, this is a viable solution.

ANTP to NTP downgrade. ANTP servers are also NTP servers, since ANTP is implemented as an NTP extension. This eases deployment; older clients can continue using NTP, while newer clients can use ANTP. However, a network adversary can drop the ANTP extension from the request, and the server will respond with NTP (having interpreted the request as NTP). For this reason, clients that send an ANTP request must only update their clock based on a valid ANTP response, and ignore NTP responses. For similar reasons, clients are not recommended to implement a fall back from ANTP to NTP.

4 Implementation and Performance

Here we describe our instantiation of ANTP in terms of cryptographic primitives used as well as its implementation and results on its performance.²

4.1 Instantiation and Implementation

We instantiate ANTP using the following cryptographic algorithms. We use AES128-GCM as the symmetric encryption algorithm for the server to encrypt and decrypt state, SHA-256 as the hash algorithm, and HMAC-SHA256 as the MAC and key derivation function. We support two key encapsulation mechanisms, RSA key transport and static-ephemeral elliptic curve Diffie-Hellman:

- RSA key transport: In KeyGen, the public key and secret key are a 2048-bit RSA key pair. Encap is defined by selecting a key $pms \leftarrow \{0, 1\}^{128}$ and encrypting pms using the RSA public key with RSA-PKCS#1.5 encryption; Decap performs decryption with the corresponding RSA secret key. The shared secret is $k \leftarrow KDF(pms, \perp, "ANTP", len)$.
- Static-ephemeral elliptic curve Diffie–Hellman: Let P be the generator (base point) of the NIST-P256 elliptic curve group of prime order q . In KeyGen, the secret key is $sk \leftarrow \mathbb{Z}_q$ and the public key is $pk = sk \cdot P$, where \cdot denotes scalar-point multiplication. In Encap, select $r \leftarrow \mathbb{Z}_q$ and compute $c \leftarrow r \cdot P$ and $pms \leftarrow X(r \cdot pk)$, where $X(Q)$ gives the x-coordinate of elliptic curve point Q . In Decap, compute $pms \leftarrow X(sk \cdot c)$. The shared secret is $k \leftarrow KDF(pms, \perp, "ANTP" \| c, len)$. This is the ECIES-KEM [29] which is IND-CCA secure under the el-

²Our implementation code and benchmarking tools can be found at <https://github.com/DowlingBJ/AuthenticatedNTP>.

liptic curve discrete logarithm assumption in the random oracle model [8].

We implemented ANTP by extending OpenNTPD version 1.92 [34]. Our implementation relies on OpenSSL version 1.0.2f [35] for its cryptographic components; notably, this version includes a high-speed assembly implementation of the NIST P-256 curve; AES-GCM encryption/decryption uses the AES-NI instruction.

4.2 Performance

Methodology. We collected performance measurements for each of the negotiation, key exchange, and time synchronization phases. We wanted to know the maximum number of connections per second that could be supported in each phase, as well as the latency a client would experience for a typical server. For comparison we also collected performance measurements for unauthenticated NTP time synchronization phases.

Our throughput and LAN latency experiments were carried out in the following environment on our local area network. We had two machines acting as clients, and a single server machine running ANTP. The server had an Intel Core i7-4770 (Haswell) processor with 4 cores running at 3.4 GHz with 15.6 GiB of RAM; we used two similar client machines, which in our experiments were always sufficient to saturate the server. The clients and server were connected over an isolated 1 gigabit LAN. The server was running Linux Mint 17.2 with no other software installed.

Our latency experiments across the US were carried out between two Amazon AWS m4.2xlarge instances, the server in the US East (N. Virginia) region and the client in the US West 1 (N. California) region. These instances each had eight virtual CPUs, each of which was an Intel Xeon E5-2676 v3 (Haswell) core running at 2.4 GHz, with 32 GiB of RAM, and 1 Gbps of dedicated bandwidth; the instances were running Ubuntu 14.04 with no other software installed.

It is important to note that OpenNTPD is not multi-threaded, so the OpenNTPD server process runs on a single core, regardless of the number of cores on the machine. As the key exchange phase is CPU bound, in a threaded server implementation we expect key exchange phase throughput to increase linearly with the number of CPU cores until bandwidth is saturated.

For testing throughput (connections/second), we used our own multi-threaded UDP flooding benchmarking tool that sends static packets and collects the number of responses, the average latency of those responses, and the number of dropped packets. We tuned the number of queries per second to ensure that the server’s (single) core had around 95% utilization, and that more client packets were sent than being processed, but not so many more that

performance became degraded (i.e., the server dropped less than 1% of packets being received per second).

For testing individual phase latency, we again used our UDP benchmarking tool, this time measuring latency of a subset of connections while maintaining a particular background ANTP load at the server (either 50% or 90% of supported throughput), to measure the latency a client would experience at an unloaded or loaded server.

For testing total protocol runtime, we instrumented the OpenNTPD client to report the runtime of a single complete (all three phases) ANTP synchronization, again with background ANTP load as above.

Results – individual phases.

Table 1 shows the results of each phase. Results reported are the average of 5 trials to prevent outlier results. For throughput and individual phase latency, each trial was run for 100 seconds. For throughput, Table 1 reports the number of response packets received at the client machine.

Negotiation phases. The lower throughput of RSA and ECDH negotiation messages (compared to NTP) is due to larger message size of ANTP messages, as network bandwidth was saturated for this measurement. Latency for ECDH negotiation at 90% load is higher compared to RSA negotiation at 90% load; at that load level, a much larger number of ECDH packets are being sent than RSA packets, so CPU load in the ECDH is higher even though they have the same bandwidth consumption, leading to higher latency for ECDH negotiation.

Key exchange phases. As expected, server key exchange throughput is higher when ECC is used for public key operations compared to RSA. This difference is explained by the relative costs of the underlying cryptographic operations: using OpenSSL’s speed command for benchmarking individual crypto operations, the runtime of ECC NIST P-256 point multiplication is $8.62 \times$ faster than RSA 2048 private key operations, whereas we observe a $7.54 \times$ improvement in throughput for ANTP’s ECDH key exchange over ANTP’s RSA key exchange. Latency on the local network for RSA key exchange is approximately $2.9 \times$ that of ECDH key exchange at 90% load.

Time synchronization phases. While ANTP time synchronization phases are more computationally intensive than unauthenticated NTP, throughput is reduced by only a factor of approximately 1.6. Since this phase is CPU bound, we expect a multi-threaded server implementation to increase ANTP throughput. Latency increase on the local network for ANTP at 50% load is only about 14% and at 90% load is about 27%.

Results and extrapolation – all 3 phases. Since each client makes a full 3-phase time synchronization (negotiation, followed by key exchange, followed by time synchronization) relatively infrequently, it does not make

sense to measure server throughput for complete 3-phase time synchronizations. We did measure latency of a 3-phase time synchronization to note the performance that a client would perceive on its initial synchronization. As expected, the total runtime of a client exceeds the sum of the latencies from each individual phase due to the client performing its own cryptographic operations.

It is interesting to note that latency slows as the server approaches load capacity. Future work on OpenNTPD and other NTP servers could include optimizations to reduce latency and improve time synchronization accuracy under increasing load.

We can extrapolate from the individual phase results the client pool that ANTP could feasibly support running on the same hardware. For example, Windows by default polls time servers every 9 hours [13]. Assuming this is true for all clients (and that the clients synchronize uniformly across the period) 175,644 time synchronization requests per second would correspond to a pool of 5,755,502,592 clients.

ANTP clients would choose how often to restart the negotiation phase and we recommend doing so periodically to ensure the attack window from exposure of the symmetric key is limited. If keys are re-exchanged monthly, this is a ratio of 1:1:1440 for expected negotiation, key exchange, and time synchronization messages, which increases to 1:1:8640 if clients re-exchanged every 6 months. From these or other expected ratios, one could extrapolate the expected performance impact of using ANTP over NTP.

5 Security Framework

In this section we introduce our new time synchronization provable security framework for analyzing time synchronization protocols such as ANTP, NTP, and the Precision Time Protocol. It builds on both the Bellare–Rogaway model [5] for *authenticated key exchange* and the Jager et al. framework for *authenticated and confidential channel establishment* [11]. Neither of those models however includes time.

Our framework models time as a counter that each party separately maintains, as the goal of the protocol is to synchronize these disparate counters. Additionally, the adversary in our execution environment has the ability to initialize each protocol run with a new time counter independent of the party’s own counter, and controls when protocol runs can increment their counter, effectively giving the adversary complete control of both the latency of the network and the computation time of the parties.

5.1 Execution Environment

There are n_p parties P_1, \dots, P_{n_p} , each of whom is a protocol participant. Each party generates a long-term key-pair

(sk_i, pk_i) , and can run up to n_s instances of the protocol which are referred to as sessions. We denote the s th session of a party P_i as π_i^s . Note that each session π_i^s has access to the long-term key pair of the party P_i . In addition, we denote with T and T_c the full transcript and server-session maintained client transcript T_c .

Per-Session Variables. The following variables are maintained by each session:

- $\rho \in \{\text{client}, \text{server}\}$: the role of the party.
- $id \in \{1, \dots, n_p\}$: the identity of the party.
- $pid \in \{1, \dots, n_p\}$: the believed identity of the partner.
- $\alpha \in \{\text{accept}, \text{reject}, \text{in-progress}\}$: the session status.
- $k \in \{0, 1\}^{128}$: the session key.
- $T_c \in \{\{0, 1\}^*, \emptyset\}$: if $\rho = \text{server}$, the transcript of client messages, otherwise $T_c = \emptyset$.
- $T \in \{0, 1\}^*$: the transcript of messages sent and received.
- $time \in \mathbb{N}$: a counter maintained by the session.

Adversary Interaction. The adversary schedules and controls all interactions between protocol participants. The adversary has control of all communication, able to create, delete, reorder or modify messages at will. The adversary can compromise long-term and session keys. Additionally, the adversary is able to set the clock of a party to an arbitrary time when a session begins and control the rate at which time progresses during the execution of a session. The following queries model normal execution with adversary control of time:

- $\text{Create}(i, r, t)$: The adversary activates a new session with party P_i , initializing $\pi_i^s.\rho = r$ and $\pi_i^s.time = t$. Note that if $\pi_i^s.\rho = \text{client}$, then π_i^s responds with the first message of the protocol run.
- $\text{Send}(i, s, m, \vec{\Delta})$: The adversary sends a message m to a session π_i^s . Party P_i processes the message m and responds according to protocol specification, updating per-session variables and outputting some message m^* if necessary. During message processing, the party may execute multiple calls to a distinguished $\text{Now}()$ procedure, modelling the party reading its current time from memory; immediately before the ℓ th such call to the $\text{Now}()$ procedure, the session’s $\pi_i^s.time$ variable is incremented by Δ_ℓ .

These queries model compromise of secret data:

- $\text{Reveal}(i, s)$: The adversary receives the session key k of the session π_i^s .
- $\text{Corrupt}(i)$: The adversary receives the long-term secret-key sk_i of the party P_i .

The following query allows additional adversary control of the clock:

- $\text{Tick}(i, s, \Delta)$: The adversary increments the counter $\pi_i^s.\text{time}$ by Δ .

The vector $\vec{\Delta}$ in Send is necessary due to subtleties in the security framework: An adversary cannot issue Tick queries to a session during the processing of a Send query, but a party may read its clock multiple times while processing a message and thus expect to receive different clock times. The vector $\vec{\Delta}$ in the Send query allows adversary control of this clock rate.

Note that our model assumes that during execution of a session, the clocks between two parties advance at the same rate, otherwise it does not make sense for two parties to try to synchronize their clocks at all. This implicitly assumes that the parties are in the same reference frame. Additionally, while computer clocks may progress at different rates, we are assuming that, over a relatively short period of time, like the few seconds for an execution of the protocol, the difference in clock rate will be negligible. This will be formalized in Definitions 3 and 4 with the condition that the adversary advances the *time* of matching sessions symmetrically: a $\text{Tick}(j, t, \sum_{l=1}^{\ell} \Delta_l)$ must be issued if session π_j^t matching π_i^s exists when $\text{Send}(i, s, m, \vec{\Delta})$ is issued.

Security Experiment. The *time synchronization security experiment* is played between a challenger \mathcal{C} who implements all n_p parties according to the execution environment and protocol specification, and an adversary \mathcal{A} . After the challenger generates the long-term key pairs, the adversary receives the list of public keys and interacts with the challenger using the queries described above. Eventually the adversary terminates.

5.2 Security Definitions

The goal of the adversary, formalized in this section, is to break time synchronization security by causing any client session to complete a session with a time counter such that $|\pi_i^s.\text{time} - \pi_j^t.\text{time}| > \delta$, (where π_j^t is the partner of the session π_i^s such that $\pi_j^t.id = \pi_i^s.pid$, and δ is an accuracy margin) or cause a session π_i^s to accept a protocol run without having a matching session π_j^t . The adversary controls the initialization of the party's clock in each session, and the rate at which the clock advances during each session, with the restriction that during execution of a session the adversary must advance the party and its peer at the same rate.

5.2.1 Matching Conversations and Authentication

Authentication is defined similarly to the approach of Bellare and Rogaway [5], by use of matching conversations. We use the variant of matching conversations employed by Jager et al. [11], and modify the definition to reflect client authentication of stateless servers.

Definition 1 (Matching Conversations). *We say a session π_i^s matches a session π_j^t if $\pi_i^s.\rho \neq \pi_j^t.\rho$ and $\pi_i^s.T$ prefix-matches $\pi_j^t.T$. For two transcripts T and T' , we say that T is a prefix of T' if $|T| \neq 0$ and T' is identical to T for the first $|T|$ messages in T' . Two transcripts T and T' prefix-match if T is a prefix of T' , or T' is a prefix of T .*

Prefix-matching prevents an adversary from trivially winning the experiment by dropping the last protocol message after a session has accepted. Note that since our focus is clients authenticating stateless servers, our authentication definition is one-sided.

Definition 2 (Stateless Server Authentication). *We say that a session π_i^s accepts maliciously if:*

- $\pi_i^s.\alpha = \text{accept}$;
- $\pi_i^s.\rho = \text{client}$;
- no $\text{Reveal}(i, s)$ or $\text{Reveal}(j, t)$ queries were issued before $\pi_i^s.\alpha \leftarrow \text{accept}$ and π_j^t matches π_i^s ;
- no $\text{Reveal}(j, t')$ queries were issued before $\pi_i^s.\alpha \leftarrow \text{accept}$ and $\pi_j^{t'}.T_c = \pi_j^t.T_c$;
- no $\text{Corrupt}(j)$ query was ever issued before $\pi_i^s.\alpha \leftarrow \text{accept}$, where $j = \pi_i^s.pid$;

but there exists no session π_j^t such that π_i^s matches π_j^t .

We define $\text{Adv}_{\mathcal{T}}^{\text{auth}}(\mathcal{A})$ as the probability of \mathcal{A} forcing any session π_i^s of time synchronization protocol \mathcal{T} to accept maliciously.

In the above definition, the first Reveal condition prevents \mathcal{A} from trivially winning the experiment by accessing the session key of the Test session. Similarly the Corrupt condition prevents \mathcal{A} from trivially winning by decrypting the premaster secret with the session partner's public key. The possibility exists for an adversary to trivially win the experiment by replaying client messages to a second session and querying the second session with Reveal. Disallowing Reveal queries in general is clearly too restrictive, so we prevent this in the second Reveal condition by disallowing Reveal queries to server sessions with matching client transcripts.

5.2.2 Correct and Secure Time Synchronization

The goal of a time synchronization protocol is to ensure that the difference between the two parties' clocks is within a specified bound. A protocol is δ -correct if that difference can be bounded in honest executions of the protocol, and δ -accurate secure if that difference can be bounded even in the presence of an adversary.

Definition 3 (δ -Correctness). *A protocol \mathcal{T} satisfies δ -correctness if, in the presence of a passive adversary that faithfully delivers all messages and increments in each partner session symmetrically, then the client and server's*

clocks are within δ of each other. More precisely, in the presence of a passive adversary, for all sessions π_i^s where

- $\pi_i^s.\alpha = \text{accept}$;
- $\pi_i^s.\rho = \text{client}$;
- whenever \mathcal{A} queries $\text{Send}(i, s, m, \vec{\Delta})$ or $\text{Send}(j, t, m', \vec{\Delta}')$, \mathcal{A} also queries $\text{Tick}(j, t, \sum_{i=1}^{\ell} \Delta_i)$ or $\text{Tick}(i, s, \sum_{i=1}^{\ell} \Delta'_i)$, respectively; and
- whenever \mathcal{A} queries $\text{Tick}(i, s, \Delta)$, or $\text{Tick}(j, t, \Delta')$, \mathcal{A} also queries $\text{Tick}(j, t, \Delta)$ or $\text{Tick}(j, t, \Delta')$, respectively;

we must also have that $|\pi_i^s.\text{time} - \pi_j^t.\text{time}| \leq \delta$.

Definition 4 (δ -Accurate Secure Time Synchronization). We say that an adversary \mathcal{A} breaks the δ -accuracy of a time synchronization protocol if when \mathcal{A} terminates, there exists a session π_i^s with partner id $\pi_i^s.\text{pid} = j$ such that:

- $\pi_i^s.\alpha = \text{accept}$;
- $\pi_i^s.\rho = \text{client}$
- \mathcal{A} made no $\text{Corrupt}(j)$ query before $\pi_i^s.\alpha \leftarrow \text{accept}$;
- \mathcal{A} made no $\text{Reveal}(i, s)$ or $\text{Reveal}(j, t)$ query before $\pi_i^s.\alpha \leftarrow \text{accept}$ and π_j^t matches π_i^s ;
- while $\pi_i^s.\alpha = \text{in-progress}$ and \mathcal{A} queried $\text{Send}(i, s, m, \vec{\Delta})$ or $\text{Send}(j, t, m', \vec{\Delta}')$ (where π_i^s matches π_j^t), then \mathcal{A} also queried $\text{Tick}(j, t, \sum_{i=1}^{\ell} \Delta_i)$ or $\text{Tick}(i, s, \sum_{i=1}^{\ell} \Delta'_i)$, respectively;
- while $\pi_i^s.\alpha = \text{in-progress}$ and \mathcal{A} queried $\text{Tick}(i, s, \Delta)$, or $\text{Tick}(j, t, \Delta')$ (where π_i^s matches π_j^t), then \mathcal{A} also queried $\text{Tick}(j, t, \Delta)$ or $\text{Tick}(i, s, \Delta')$, respectively; and
- $|\pi_i^s.\text{time} - \pi_j^t.\text{time}| > \delta$.

The probability an adversary \mathcal{A} has in breaking δ -accuracy of a time synchronization protocol \mathcal{T} is denoted $\text{Adv}_{\mathcal{T}, \delta}^{\text{time}}(\mathcal{A})$.

5.3 Multi-Phase Protocols

Our construction in Section 3 has a single run of the negotiation and key exchange phases, followed by multiple time synchronization executions reusing the negotiated cryptographic algorithms and shared secret key. To model the security of such *multi-phase* time synchronization protocols, we further extend our framework so that a single session can include multiple time synchronization phases. The differences from the model described in the previous section are detailed below.

Per-Session Variables. The following variables are added or changed:

- $n \in \mathbb{N}$: the number of time synchronization phases allowed in this session.

- time_p , for $p \in \{1, \dots, n\}$: the time recorded at the conclusion of phase p .

- $\alpha \in \{\text{accept}_p, \text{reject}_p, \text{in-progress}_p\}$, for $p \in \{1, \dots, n\}$: the status of the session. Note that, when phase p concludes and $\alpha \leftarrow \text{accept}_p$ is set, the party also sets $\text{time}_p \leftarrow \text{time}$.

Adversary Interaction. The adversary can direct the client to run an additional time synchronization phase with a new Resync query, and the client will respond according the protocol specification. The Create query in this setting is also changed:

- $\text{Create}(i, r, t, n)$: Proceeds as for $\text{Create}(i, r, t)$, and also sets $\pi_i^s.n \leftarrow n$.
- $\text{Resync}(i, s, \vec{\Delta})$ - The adversary indicates to a session π_i^s to begin the next time synchronization phase. Party P_i responds according to protocol specification, updating per-session variables and outputting some message m^* if necessary. During message processing, immediately before the ℓ th call to the $\text{Now}()$ procedure, the session's $\pi_i^s.\text{time}$ variable is incremented by Δ_ℓ .

The goal of the adversary is also slightly different to account for the possibility of breaking time synchronization of any given time synchronization phase: the adversary's goal is to cause a client session to have *any* phase where its time is desynchronized from the server's. In particular, for there to be some client instance π_i^s and some phase p such that $|\pi_i^s.\text{time}_p - \pi_j^t.\text{time}_p| > \delta$ where π_j^t is the partner of session π_i^s . Again the adversary in general controls clock ticks and can tick parties at different rates, however must tick clocks at the same rate when phases have switched back to being in-progress.

Definition 5 (δ -Accurate Secure Multi-Phase Time Synchronization). We say that an adversary \mathcal{A} breaks the δ -accuracy of a multi-phase time synchronization protocol if when \mathcal{A} terminates, there exists a phase p session π_i^s with partner id $\pi_i^s.\text{pid} = j$ such that:

- $\pi_i^s.\rho = \text{client}$
- $\pi_i^s.\alpha = \text{accept}_q$ for some $q \geq p$;
- \mathcal{A} did not make a $\text{Corrupt}(j)$ query before $\pi_i^s.\alpha \leftarrow \text{accept}_p$ was set;
- \mathcal{A} did not make a $\text{Reveal}(i, s)$ or $\text{Reveal}(j, t)$ query before $\pi_i^s.\alpha \leftarrow \text{accept}_p$ was set and π_j^t matches π_i^s ;
- while $\pi_i^s.\alpha = \text{in-progress}$ and \mathcal{A} queried $\text{Send}(i, s, m, \vec{\Delta})$ or $\text{Send}(j, t, m', \vec{\Delta}')$, then \mathcal{A} also queried $\text{Tick}(j, t, \sum_{i=1}^{\ell} \Delta_i)$ or $\text{Tick}(i, s, \sum_{i=1}^{\ell} \Delta'_i)$, respectively;
- while $\pi_i^s.\alpha = \text{in-progress}$ and \mathcal{A} queried $\text{Tick}(i, s, \Delta)$, or $\text{Tick}(j, t, \Delta')$, then \mathcal{A} also queried $\text{Tick}(j, t, \Delta)$ or $\text{Tick}(i, s, \Delta')$, respectively; and
- $|\pi_i^s.\text{time}_p - \pi_j^t.\text{time}_p| > \delta$.

The probability an adversary \mathcal{A} has in breaking δ -accuracy of multi-phase time synchronization protocol \mathcal{T} is denoted $\text{Adv}_{\mathcal{T}, \delta}^{\text{multi-time}}(\mathcal{A})$.

6 Security of ANTP

Here we present ANTP correctness and security theorems.

6.1 Correctness

Theorem 1 (Correctness of ANTP). *Fix $E \in \mathbb{N}$. ANTP_E is an E -correct time synchronization protocol as defined in Definition 3.*

Proof. When analyzing ANTP in terms of correctness, we can restrict analysis to data that enters the clock-update procedure as input, as the rest of the protocol is designed to ensure authentication and does not influence the session’s *time* counter. This allows us to narrow our focus to SNTP, which is the time synchronization core of ANTP.

We first focus on a single time synchronization phase. At the beginning of the time synchronization phase of ANTP, the client will send an NTP request (`req`) which contains t_1 , the time the client sent `req`. Note that the adversary is restricted to delivering the messages faithfully as a passive adversary, and also must increment the time of each protocol participant symmetrically. The adversary otherwise has complete control over the passage of time. Thus $\theta_1, \theta_2, \theta_3$ are non-negative but otherwise arbitrary values selected by the adversary (where θ_1 is the propagation time from client to server, θ_2 is server processing time and θ_3 is propagation time from server to client). Thus the client computes the round-trip time of the protocol as: $RTT = (t_4 - t_1) - (t_3 - t_2) = \theta_1 + \theta_3$ and approximates the server-to-client propagation time as $\tilde{\theta}_3 = \frac{1}{2}(\theta_1 + \theta_3)$.

When the client-to-server and server-to-client propagation times are equal ($\theta_1 = \theta_3$) then $\tilde{\theta}_3 = \theta_3$, and the values t_3 and t_2 allow the client to exactly account for θ_2 . The *time* counter is updated by $\text{time} + \text{offset} = t_3 + \tilde{\theta}_3 - t_4$, and upon completion the client’s clock is exactly synchronized with the server’s clock.

When $\theta_1 \neq \theta_3$, we have that $\theta_3 - \tilde{\theta}_3 = \frac{1}{2}(\theta_3 - \theta_1)$, so the statistics t_1, \dots, t_4 do not allow the client to exactly account for client-to-server propagation time θ_3 ; the client’s updated time may be off by up to $\frac{1}{2}(\theta_3 - \theta_1)$. Fortunately, we can bound this value by E : we know that $\frac{1}{2}(\theta_3 - \theta_1) \leq \frac{1}{2}(\theta_1 + \theta_3)$, and furthermore we know that ANTP_E will only accept time synchronization when $\frac{1}{2}(\theta_1 + \theta_3) \leq E$, so in sessions that accept (assuming a passive adversary) we have that the client’s clock is at most $\frac{1}{2}(\theta_3 - \theta_1) \leq E$ different from the server’s clock.

Now moving to the multi-phase setting, we note that this analysis of the correctness of ANTP applies to each

separate time synchronization phase: since the client’s (t_1, t_4) values are only used to calculate the total round-trip time of the time synchronization phase, thus if the rate-of-time for both client and server during the phase is the same, each phase is also E -accurate in the presence of a passive adversary, even if the adversary dramatically changes the rate-of-time for partners between time synchronization phases. \square

6.2 Security

Security of a single 3-phase execution of ANTP in the sense of Definition 4 is given by Theorem 2 below. Security of multiple phases in the sense of Definition 5 follows with a straightforward adaptation.

Intuitively, the bound on the possible error that an \mathcal{A} can introduce without altering packets is as in Section 3. It follows then that if all messages are securely authenticated, and the only inputs to the clock-update procedure are either: authenticated via messages, or the round trip delay RTT ; then any attacker can only introduce at most E error into the clock-update procedure (where $E \geq RTT$).

Theorem 2 (Security of ANTP). *Fix $E \in \mathbb{N}$ and let λ be the length of the nonces in m_1 and m_5 (in our instantiation, $\lambda = 256$). Assuming the key encapsulation mechanism KEM (with keyspace $\mathcal{KEM.K}$) is IND-CCA-secure, the message authentication code MAC is EUF-CMA-secure, the hash function Hash is collision-resistant, and the key derivation function KDF and authenticated encryption scheme AE are secure, then ANTP_E is a E -accurate secure time synchronization protocol as in Definition 4. In particular, there exist algorithms $\mathcal{B}_3, \dots, \mathcal{B}_8$, described in the proof of the theorem, such that, for all adversaries \mathcal{A} , we have*

$$\begin{aligned} \text{Adv}_{\text{ANTP}_E, E}^{\text{time}}(\mathcal{A}) &\leq \frac{n_p^2 n_s^2}{2^{\lambda-2}} + n_p^2 n_s^2 \left(\text{Adv}_{\text{Hash}}^{\text{coll}}(\mathcal{B}_3^{\mathcal{A}}) \right. \\ &\quad + \text{Adv}_{\text{AE}}^{\text{auth-enc}}(\mathcal{B}_4^{\mathcal{A}}) + \text{Adv}_{\text{KEM}}^{\text{ind-cca}}(\mathcal{B}_5^{\mathcal{A}}) \\ &\quad + \text{Adv}_{\text{KDF}}^{\text{kdf}}(\mathcal{B}_6^{\mathcal{A}}) + \text{Adv}_{\text{AE}}^{\text{auth-enc}}(\mathcal{B}_7^{\mathcal{A}}) \\ &\quad \left. + \text{Adv}_{\text{MAC}}^{\text{euf-cma}}(\mathcal{B}_8^{\mathcal{A}}) \right) \end{aligned}$$

where n_p and n_s are the number of parties and sessions created by \mathcal{A} during the experiment.

The standard definitions for security of the underlying primitives and the corresponding advantages $\text{Adv}_{\text{AE}}^{\text{auth-enc}}(\mathcal{A})$, $\text{Adv}_{\text{KEM}}^{\text{ind-cca}}(\mathcal{A})$, $\text{Adv}_{\text{Hash}}^{\text{coll}}(\mathcal{A})$, $\text{Adv}_{\text{MAC}}^{\text{euf-cma}}(\mathcal{A})$, and $\text{Adv}_{\text{KDF}}^{\text{kdf}}(\mathcal{A})$ are given in the full version [6].

Proof. From Theorem 1, ANTP_E is an E -correct time synchronization protocol in the sense of Definition 3. Thus all passive adversaries have probability 0 of breaking E -accuracy of ANTP_E . If we show that the advantage

$\text{Adv}_{\text{ANTP}_E}^{\text{auth}}(\mathcal{A})$ of any adversary \mathcal{A} of breaking authentication security (i.e., to accept without session matching) of ANTP_E is small, then it follows that the advantage of any active adversary \mathcal{A} in breaking E -accuracy of ANTP_E is similarly small. In other words, it immediately is the case that $\text{Adv}_{\text{ANTP}_E, E}^{\text{time}}(\mathcal{A}) \leq \text{Adv}_{\text{ANTP}_E}^{\text{auth}}(\mathcal{A})$.

We now focus on bounding $\text{Adv}_{\text{ANTP}_E}^{\text{auth}}(\mathcal{A})$. In order to show that an active adversary has negligible probability in breaking ANTP_E authentication, we use a proof structured as a sequence of games. We let $\Pr(break_i)$ denote the probability that the adversary causes some session to accept maliciously in game i . We iteratively change the security experiment, and demonstrate that the changes are either failure events with negligible probability of occurring or that if the changes are distinguishable we can construct an adversary capable of breaking an underlying cryptographic assumption. Since the client will only accept synchronization if all three phases are properly authenticated, the advantage of an active adversary is negligible given our cryptographic assumptions.

Game 0. This is the original time synchronization game described in § 4: $\text{Adv}_{\text{ANTP}_E}^{\text{auth}}(\mathcal{A}) = \Pr(break_0)$.

Game 1. In this game, we abort the simulation if any nonce is used in two different sessions by client instances. There are at most $2n_s n_p$ nonces used by client instances, each λ bits. The probability that a collision occurs among these values is $(2n_s n_p)^2 / 2^\lambda$, so:

$$\Pr(break_0) \leq \Pr(break_1) + \frac{n_s^2 n_p^2}{2^{\lambda-2}}$$

Game 2. Here, we guess the first client session to accept maliciously, aborting if incorrect. We select randomly from two indices $(i, s) \leftarrow \$_{\{1, \dots, n_p\} \times \{1, \dots, n_s\}}$ and abort if π_i^s is not the first session to accept maliciously. Now the challenger responds to $\text{Reveal}(i, s)$ queries (if $\pi_i^s.\alpha = \text{accept}$) by aborting the game, as it follows that the guessed session cannot accept maliciously. There are at most $n_p n_s$ client sessions, and we guess the first session to accept maliciously with probability at least $1/n_p n_s$, so $\Pr(break_1) \leq n_p n_s \Pr(break_2)$.

Game 3. Here we guess the partner session to π_i^s , by selecting from two indices $(j, t) \leftarrow \$_{\{1, \dots, n_p\} \times \{1, \dots, n_s\}}$ and abort if π_j^t is not the partner session to π_i^s . Now, the challenger answers $\text{Corrupt}(j)$ and $\text{Reveal}(j, t)$ queries before $\pi_i^s.\alpha \leftarrow \text{accept}$ by aborting the game, as it follows that the guessed session cannot accept maliciously. There are at most $n_p n_s$ server sessions, and we guess the partner of the first session to accept maliciously with probability at least $1/n_p n_s$, so $\Pr(break_2) \leq n_p n_s \Pr(break_3)$.

Game 4. Here we abort if a hash collision occurs, by computing all hash values honestly and aborting if there exists two evaluations $(in, \text{Hash}(in)), (\hat{in}, \text{Hash}(\hat{in}))$ such that $in \neq \hat{in}$ but $\text{Hash}(in) = \text{Hash}(\hat{in})$. The simulator interacts with a Hash-collision challenger, outputting

the collision if found. Thus: $\Pr(break_3) \leq \Pr(break_4) + \text{Adv}_{\text{Hash}}^{\text{coll}}(\mathcal{B}_3^A)$.

Game 5. In this game, we abort if in server session π_j^t the ciphertext received in m_3 is not equal to the ciphertext sent in m_1 but the output of AuthDec_s is not \perp .

We construct an algorithm \mathcal{B}_4^A that simulates Game 4 identically, except to interact with an AE challenger in the following way: When P_j needs to run AuthEnc or AuthDec , \mathcal{B}_4^A uses its oracles to compute the required value. In server session π_j^t , when \mathcal{B}_4^A receives a ciphertext in m_3 that was not equal to the ciphertext sent in m_1 but the output of the AuthDec oracle is not \perp , this corresponds to a ciphertext forgery, and thus: $\Pr(break_4) \leq \Pr(break_5) + \text{Adv}_{\text{AE}}^{\text{auth-enc}}(\mathcal{B}_4^A)$.

Game 6. In this game, sessions π_i^s and π_j^t compute the session key k by applying KDF to a random secret $pms' \leftarrow \$_{\text{KEM.K}}$, rather than the pms that was encapsulated using KEM.Encap and transmitted in ciphertext e . Any algorithm used to distinguish Game 5 from Game 6 can be used to construct an algorithm capable of distinguishing KEM encrypted values via plaintext, thus breaking IND-CCA security of the key encapsulation mechanism.

We construct a simulator \mathcal{B}_5^A that interacts with a KEM challenger. \mathcal{B}_5^A activates party P_j with the public key pk received from the challenger. \mathcal{B}_5^A responds identically to queries from \mathcal{A} as in Game 5, except as follows:

- \mathcal{B}_5^A computes the KEM ciphertext e for the session π_i^s by obtaining a challenge (e, pms) from its KEM challenger.
- \mathcal{B}_5^A computes $\pi_i^s.k \leftarrow \text{KDF}(pms, \dots)$
- In any P_j session where m_3 contains the challenge ciphertext above, \mathcal{B}_5^A computes the session key as $k \leftarrow \text{KDF}(pms, \dots)$.
- In any other P_j session where m_3 does not contain the challenge ciphertext above, \mathcal{B}_5^A queries the ciphertext to its Decap oracle to obtain the premaster secret and uses that as its input to KDF to compute the session key k .
- \mathcal{B}_5^A never needs to answer a $\text{Corrupt}(j)$ query because of Game 3.

When the random bit b sampled by the KEM ind-cca challenger is 0, pms is truly the decapsulation of the ciphertext e , in which case \mathcal{B}_5^A perfectly simulates of Game 5. When $b = 1$, pms is random and independent of e , in which case \mathcal{B}_5^A perfectly simulates Game 6. Observe that \mathcal{B}_5^A never asks the challenge ciphertext e to its decapsulation oracle.

An adversary capable of distinguishing Game 5 from Game 6 can therefore be used to break IND-CCA security of KEM, so $\Pr(break_5) \leq \Pr(break_6) + \text{Adv}_{\text{KEM}}^{\text{ind-cca}}(\mathcal{B}_5^A)$.

Game 7. In this game, we replace the secret key k in sessions π_i^s and π_j^t with a uniformly random value k' from

$\{0,1\}^{l_{\text{KDF}}}$ where l_{KDF} is the length of the KDF output, instead of being computed honestly via $k \leftarrow \text{KDF}(pms, \dots)$.

In Game 6, we replaced the premaster secret value pms with a uniformly random value from $\text{KEM}.\mathcal{K}$. Thus, any algorithm that can distinguish Game 6 from Game 7 can distinguish the output of KDF from random. We explicitly construct such a simulator \mathcal{B}_6^A that interacts with a KDF challenger, and proceeds identically to Game 6, except: when computing k for π_i^s , \mathcal{B}_6^A queries the KDF challenger with pms ; and when computing k for π_j^t , \mathcal{B}_6^A sets $\pi_j^t.k = \pi_i^s.k$. When the random bit b sampled by the KDF challenger is 0, $k = \text{KDF}(pms, \dots)$, and \mathcal{B}_6^A provides a perfect simulation of Game 6. When $b = 1$, $k \leftarrow \{0,1\}^{l_{\text{KDF}}}$ and \mathcal{B}_6^A provides a perfect simulation of Game 7.

An adversary capable of distinguishing Game 6 from Game 7 can therefore distinguish the output of KDF from random, so $\Pr(\text{break}_6) \leq \Pr(\text{break}_7) + \text{Adv}_{\text{KDF}}^{\text{kdf}}(\mathcal{B}_6^A)$.

Game 8. In this game, in session π_j^t we replace the contents of the ciphertext C_2 sent in m_3 with a random string of the same length, and abort if the ciphertext received in m_5 is not equal to the ciphertext sent in m_3 but the output of the AuthDec_s algorithm is not \perp .

We construct an algorithm \mathcal{B}_7^A that interacts with an AE challenger in the following way: \mathcal{B}_7^A acts exactly as in game 7 except for sessions run by party P_j . In session π_j^t , for the computation of C_2 , \mathcal{B}_7^A picks a uniformly random binary string z' of length equal to $z = k\|\text{KDF}\|\text{Hash}\|\text{KEM}\|\text{MAC}$ and submits (z, z') to its AuthEnc oracle. For all other computations that P_j involving AuthEnc_s or AuthDec_s , \mathcal{B}_7^A submits the query its respective AuthEnc or AuthDec oracle.

When the random bit b sampled by the AE challenger is 0, C_2 contains the encryption of z , so \mathcal{B}_7^A provides a perfect simulation of Game 7. When $b = 1$, C_2 contains the encryption of z' , so \mathcal{B}_7^A provides a perfect simulation of Game 8. An adversary capable of distinguishing Game 7 from Game 8 can therefore break the confidentiality of AE and guess b . Additionally, if \mathcal{B}_7^A receives a ciphertext in m_5 that was not equal to the ciphertext sent in m_3 but the output of the AuthDec oracle is not \perp , this corresponds to a ciphertext forgery, and thus \mathcal{B}_7^A has broken the integrity of AE. Thus, $\Pr(\text{break}_7) \leq \Pr(\text{break}_8) + \text{Adv}_{\text{AE}}^{\text{auth-enc}}(\mathcal{B}_7^A)$.

The effect of Game 8 is that, in the target session and its partner, the key used in the MAC computations is independent of the values transmitted.

Game 9. In this game, we abort when the session π_i^s accepts maliciously. We do this by constructing a simulator \mathcal{B}_8^A that interacts with the MAC challenger, but computes τ_1 and τ_2 for π_j^t by querying $h\|m_3\|C_2$ and $m_5\|t_1\|t_2\|t_3$ to the MAC challenger. \mathcal{B}_8^A verifies MAC tags for $\pi_{i^*}^{s^*}$ by again querying $h\|m_3\|C_2$ and $m_5\|t_1\|t_2\|t_3$ to the MAC

challenger and ensuring the MAC challenger's output is equal to the tag to be verified. Note that now that the key k is substituted for the key maintained by the MAC challenger: k was already uniformly random and independent of the protocol run, and by Game 2 and Game 3, the simulator already responds to Reveal queries to π_i^s and π_j^t by aborting the security experiment. Thus these changes to the game are indistinguishable. When $\pi_i^s.\alpha \leftarrow \text{accept}$, \mathcal{B}_8^A checks P_j to see if there is a matching session. Since by Game 1 all protocol flows are unique (by unique nonces), if P_j has no matching session the adversary must have produced a valid MAC tag $\hat{\tau}_1$ or $\hat{\tau}_2$ such that $\text{MAC}(k, h\|m_3\|C_2) = \hat{\tau}_1$ or $\text{MAC}(k, m_5\|t_1\|t_2\|t_3) = \hat{\tau}_2$ and (by Game 8) the key k is uniformly random. \mathcal{B}_8^A submits the appropriate pair $(h\|m_3\|C_2, \hat{\tau}_1), (m_5\|t_1\|t_2\|t_3, \hat{\tau}_2)$ to the MAC challenger and aborts. Thus, $\Pr(\text{break}_8) \leq \Pr(\text{break}_9) + \text{Adv}_{\text{MAC}}^{\text{euf-cma}}(\mathcal{B}_8^A)$.

Analysis of Game 9. We now show that an active adversary has a probability negligibly close to 0 of forcing a client session $\pi_{i^*}^{s^*}$ to accept maliciously in Game 9.

We briefly summarize the changes in games.

1. Nonces no longer collide for honest parties. Each transcript $\pi_i^s.T$ will have unique honest matching session π_j^t .
2. Guess target session; \mathcal{C} aborts if $\text{Reveal}(i, s)$ query asked.
3. Guess partner session; \mathcal{C} aborts if $\text{Corrupt}(j)$ or $\text{Reveal}(j, t)$ query asked.
4. Hash values no longer collide for honest parties. Note h is now unique for each negotiation phase, via Game 1.
5. C_1 is not forged in session π_j^t .
6. Replace premaster secret pms in target session π_i^s with a random value, rather than key encapsulated in KEM ciphertext e . Note k is unique and computed via shared secret data.
7. Replace k with uniformly random data of same length when computing τ . Thus verification of τ in Time Synchronization and Key Exchange phases is done via a uniformly random key, independent of the protocol run.
8. C_2 is not forged in session π_j^t and contains random data.
9. MAC tags in session π_i^s are not forged.

After all of the game changes, π_i^s is a target session where: no $\text{Reveal}(i, s)$ or $\text{Reveal}(j, t)$ queries were issued before $\pi_i^s.\alpha \leftarrow \text{accept}$; no $\text{Corrupt}(j)$ query was ever issued before $\pi_i^s.\alpha \leftarrow \text{accept}$, where $\pi_i^s.pid = j$; and π_i^s only accepts if $\tau_1 = \text{MAC}(k, h\|m_3\|C_2)$ and $\tau_2 = \text{MAC}(k, m_5\|t_1\|t_2\|t_3)$. By unforgeability these tags cannot be generated by \mathcal{A} and by Game 1 the protocol flow of each session is unique. τ_1 and τ_2 verification will

thus only occur if $\pi_i^s.T = \pi_j^t.T$, as τ_1 is over all messages in the negotiation and key exchange phase, and τ_2 is over all messages in the time synchronization phase and thus π_i^s will only accept if $\pi_j^t.T$ prefix-matches $\pi_i^s.T$. Thus, no client session accepts maliciously in Game 9: $\Pr(\text{break}_9) = 0$.

Summing all of the probabilities yields the desired bound, showing that ANTP_E is a E -accurate secure time synchronization protocol. \square

6.3 Multi-Phase Security

Multi-phase security of ANTP_E can be established in a similar way to single-phase security as in the previous section, with minor changes to the games in the proof to enable guessing of the first *phase* session to accept maliciously.

Theorem 3 (Multi-Phase Security of ANTP). *Fix $E, n \in \mathbb{N}$. Under the same assumptions as in Theorem 2, ANTP_E is a E -accurate secure multi-phase time synchronization protocol as defined in Definition 5. In particular, there exist algorithms $\mathcal{B}_3, \dots, \mathcal{B}_8$ described in the proof of Theorem 2, such that, for all adversaries \mathcal{A} , we have that*

$$\begin{aligned} \text{Adv}_{\text{ANTP}_E, E}^{\text{multi-time}}(\mathcal{A}) &\leq \frac{n_p^2 n_s^2}{2^{\lambda-2}} + n_p^2 n_s^2 n \left(\text{Adv}_{\text{Hash}}^{\text{coll}}(\mathcal{B}_3^{\mathcal{A}}) \right. \\ &\quad + \text{Adv}_{\text{AE}}^{\text{auth-enc}}(\mathcal{B}_4^{\mathcal{A}}) + \text{Adv}_{\text{KEM}}^{\text{ind-cca}}(\mathcal{B}_5^{\mathcal{A}}) \\ &\quad + \text{Adv}_{\text{KDF}}^{\text{kdf}}(\mathcal{B}_6^{\mathcal{A}}) + \text{Adv}_{\text{AE}}^{\text{auth-enc}}(\mathcal{B}_7^{\mathcal{A}}) \\ &\quad \left. + \text{Adv}_{\text{MAC}}^{\text{euf-cma}}(\mathcal{B}_8^{\mathcal{A}}) \right) \end{aligned}$$

where n_p, n_s, n are the maximum number of parties, sessions and phases created by \mathcal{A} during the experiment.

Proof. The proof for Theorem 3 is identical to the proof to Theorem 2 except as follows.

A new game is inserted between Game 3 and Game 4 that guesses the first time synchronization phase $p \in \{1, \dots, n\}$ that the target session π_i^s will accept maliciously: by Theorem 2, we know that a session π_i^s will not accept maliciously for time synchronization phase $p = 1$, so by this step we know that π_i^s matches π_j^t up to and including phase $p - 1$.

We also edit the final game (MAC challenger) so that \mathcal{B} aborts if π_i^s accepts maliciously in phase p . We do this by editing the final game in the following way: When processing m_5 for π_j^t in the guessed phase p (we indicate this with m_{5p}) \mathcal{B} will also compute τ_{2p} by querying the MAC challenger with $m_{5p} \| t_{1p} \| t_{2p} \| t_{3p}$, and verifies the τ_{2p} for π_i^s by querying the MAC challenger with $m_{5p} \| t_{1p} \| t_{2p} \| t_{3p}$ and accepting only if the output from the MAC challenger matches the τ_p in m_{6p} . Following the same structure as the proof to Theorem 2, we have that k is a uniformly random key generated independently from the protocol

run and this change is indistinguishable. Verification of τ will only occur if $\pi_i^s.T = \pi_j^t.T$ up to phase p , as τ_1 is over all messages in the negotiation and key exchange phase, and τ_p is over all messages in phase p . \square

7 Discussion

In this work we introduced a new authenticated time synchronization protocol called ANTP, designed to securely synchronize the time of a client and server, using public key infrastructure. Our design allows a server to perform a single public key operation per client during the infrequently performed key exchange phase, and then use only faster symmetric key operations for each subsequent time synchronization request from that client. This efficient design means that the throughput of ANTP time synchronization phases is reduced by a factor of only $1.6\times$ compared to NTP. Our protocol has been designed such that servers sharing the same long-term secret can handle different phases of the same client for load-balancing purposes. Furthermore, the server need not even store per-client state, instead securely offloading storage of that state to the client.

ANTP is accompanied by a provable security analysis showing that it provides secure time synchronization within user-specified accuracy bounds. The analysis is carried out in a new provable security framework. A novel aspect of our new framework, when compared with the long line of work on authentication definitions, is that our framework models an adversary with the ability to control the flow of time, meaning the adversary can initialize different parties' clocks to different times, and even control the rate at which their clocks are advanced. The security framework can be used for the analysis of other time synchronization protocols such as the Network Time Security (NTS) protocol and the Precision Time Protocol (PTP).

Several interesting open problems in the area of secure time synchronization remain. Since ANTP uses public keys, it inherits problems associated with public key infrastructure, such as the dangers of certificate authority compromise. All existing time synchronization protocols that rely on public keys, including ours, need to initially validate the certificate of the time server, specifically that it is within its validity period. While nonces can be combined with OCSP responses to check freshness, this cannot completely solve the “first-boot” problem. A detailed study of denial of service attacks against secure time synchronization protocols including ANTP would also be worthwhile, giving detailed consideration to both the cost of cryptographic operations in practice and the bandwidth amplification afforded by directing protocol responses to a victim.

Acknowledgements

We thank Gleb Sechenov at the Queensland University of Technology for assistance in setting up the network for the experiments. B.D. and D.S. were supported in part by Australian Research Council (ARC) Discovery Project grant DP130104304. Part of this work performed while B.D. was an intern at Microsoft Research and while D.S. was at QUT.

References

- [1] IEEE Std 1588 for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems Networked Measurement and Control Systems. Tech. rep., IEEE Instrumentation and Measurement Society, 2008.
- [2] ADRIAN, D., BHARGAVAN, K., DURUMERIC, Z., GAUDRY, P., GREEN, M., HALDERMAN, J. A., HENINGER, N., SPRINGALL, D., THOMÉ, E., VALENTE, L., VANDERSLOOT, B., WUSTROW, E., BÉGUELIN, S. Z., AND ZIMMERMANN, P. Imperfect forward secrecy: How diffie-hellman fails in practice. In *ACM CCS 15* (Oct. 2015), I. Ray, N. Li, and C. Kruegel, Eds., ACM Press, pp. 5–17.
- [3] APPELBAUM, J. tlodate, 2015. <https://github.com/ioerror/tlodate>.
- [4] BASIN, D., CAPKUN, S., SCHALLER, P., AND SCHMIDT, B. Formal reasoning about physical properties of security protocols. *ACM Trans. Inf. Syst. Secur.* 14, 2 (Sept. 2011), 16:1–16:28.
- [5] BELLARE, M., AND ROGAWAY, P. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93* (Nov. 1993), V. Ashby, Ed., ACM Press, pp. 62–73.
- [6] DOWLING, B., STEBILA, D., AND ZAVERUCHA, G. Authenticated network time synchronization. Cryptology ePrint Archive, Report 2015/171, 2015. <http://eprint.iacr.org/2015/171>.
- [7] EVANS, C., PALMER, C., AND SLEEVII, R. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), Apr. 2015.
- [8] GALINDO, D., MARTIN, S., AND VILLAR, J. L. Evaluating elliptic curve based KEMs in the light of pairings. Cryptology ePrint Archive, Report 2004/084, 2004. <http://eprint.iacr.org/2004/084>.
- [9] HEDRICK, C. Routing Information Protocol. RFC 1058 (Historic), June 1988.
- [10] HODGES, J., JACKSON, C., AND BARTH, A. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), Nov. 2012.
- [11] JAGER, T., KOHLAR, F., SCHÄGE, S., AND SCHWENK, J. On the security of TLS-DHE in the standard model. In *CRYPTO 2012* (Aug. 2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of LNCS, Springer, Heidelberg, pp. 273–293.
- [12] MALHOTRA, A., COHEN, I. E., BRAKKE, E., , AND GOLDBERG, S. Attacking the Network Time Protocol. In *NDSS 2016* (Feb. 2016), Internet Society.
- [13] MICROSOFT CORPORATION. Windows Time Service Tools and Settings. Microsoft Developer Network, May 2012. https://msdn.microsoft.com/de-de/library/cc773263%28v=ws.10%29.aspx#w2k3tr_times_tools_uhlp.
- [14] MICROSOFT CORPORATION. [MS-W32T]: W32Time Remote Protocol. Microsoft Developer Network, May 2014. <https://msdn.microsoft.com/en-us/library/cc249627.aspx>.
- [15] MILLS, D. Network Time Protocol (NTP). RFC 958, Sept. 1985.
- [16] MILLS, D. Network Time Protocol (version 2) specification and implementation. RFC 1119 (Internet Standard), Sept. 1989.
- [17] MILLS, D. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305 (Draft Standard), Mar. 1992.
- [18] MILLS, D., MARTIN, J., BURBANK, J., AND KASCH, W. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.
- [19] MILLS, D. L. On the accuracy and stability of clocks synchronized by the network time protocol in the internet system. *ACM SIGCOMM Computer Communication Review* 20, 1 (1989), 65–75.
- [20] MIZRAHI, T. Security Requirements of Time Protocols in Packet Switched Networks. RFC 7384 (Informational), Oct. 2014.
- [21] NATIONAL INSTITUTE FOR STANDARDS AND TECHNOLOGY (NIST). The NIST Authenticated NTP Service. <http://www.nist.gov/pml/div688/grp40/auth-ntp.cfm>.
- [22] PERRIG, A., CANETTI, R., TYGAR, J., AND SONG, D. The TESLA broadcast authentication protocol. *RSA CryptoBytes* 5, Summer (2002).
- [23] RÖTTGER, S. Analysis of the NTP Autokey Protocol. Masters Thesis, Technische Universität Braunschweig, Feb. 2012. http://zero-entropy.de/autokey_analysis.pdf.
- [24] SALOWEY, J., ZHOU, H., ERONEN, P., AND TSCHOFENIG, H. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), Jan. 2008.
- [25] SANTESSON, S., MYERS, M., ANKNEY, R., MALPANI, A., GALPERIN, S., AND ADAMS, C. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard), June 2013.
- [26] SCHNORR, C.-P. Efficient identification and signatures for smart cards. In *CRYPTO'89* (Aug. 1990), G. Brassard, Ed., vol. 435 of LNCS, Springer, Heidelberg, pp. 239–252.
- [27] SCHWENK, J. Modelling time, or a step towards reduction-based security proofs for OTP and kerberos. Cryptology ePrint Archive, Report 2013/604, 2013. <http://eprint.iacr.org/2013/604>.
- [28] SELVI, J. Bypassing HTTP Strict Transport Security. In *Black Hat Europe* (2014). <https://www.blackhat.com/docs/eu-14/materials/eu-14-Selvi-Bypassing-HTTP-Strict-Transport-Security-wp.pdf>.
- [29] SHOUP, V. ISO/IEC 18033-2:2006: Information technology – security techniques – encryption algorithms – part 2: Asymmetric ciphers. Tech. rep., 2006. See also <http://shoup.net/iso/std6.pdf>.
- [30] SIBOLD, D., AND RÖTTGER, S. Analysis of NTP's Autokey Protocol, 2012. <https://www.ietf.org/proceedings/83/slides/slides-83-tictoc-1.pdf>.
- [31] SIBOLD, D., RÖTTGER, S., AND TEICHEL, K. Network Time Security. IETF Internet-Draft, Jan. 2016. <https://tools.ietf.org/html/draft-ietf-ntp-network-time-security-12>.
- [32] TEICHEL, K., SIBOLD, D., AND MILIUS, S. First Results of a Formal Analysis of the Network Time Security Specification. In *Security Standardisation Research*. Springer, 2015, pp. 218–245.
- [33] TEICHEL, K., SIBOLD, D., AND MILIUS, S. An Attack Possibility on Time Synchronization Protocols Secured with TESLA-Like Mechanisms, 2016. <https://www8.cs.fau.de/staff/milius/AttackPossibilityTimeSyncTESLA.pdf>.
- [34] THE OPENBSD PROJECT. OpenNTPD version 5.7p4, Mar. 2015. <http://www.openntpd.org/>.
- [35] THE OPENSSL PROJECT. OpenSSL version 1.0.2f, Jan. 2016. <https://www.openssl.org/>.

fTPM: A Software-only Implementation of a TPM Chip

Himanshu Raj*, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox,
Paul England, Chris Finner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon,
Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten
Microsoft

Abstract: Commodity CPU architectures, such as ARM and Intel CPUs, have started to offer trusted computing features in their CPUs aimed at displacing dedicated trusted hardware. Unfortunately, these CPU architectures raise serious challenges to building trusted systems because they omit providing secure resources outside the CPU perimeter.

This paper shows how to overcome these challenges to build software systems with security guarantees similar to those of dedicated trusted hardware. We present the design and implementation of a firmware-based TPM 2.0 (fTPM) leveraging ARM TrustZone. Our fTPM is the reference implementation of a TPM 2.0 used in millions of mobile devices. We also describe a set of mechanisms needed for the fTPM that can be useful for building more sophisticated trusted applications beyond just a TPM.

1 Introduction

In recent years, commodity CPU architectures have started to offer built-in features for trusted computing. TrustZone on ARM [1] and Software Guard Extensions (SGX) [25] on Intel CPUs offer runtime environments strongly isolated from the rest of the platform’s software, including the OS, applications, and firmware. With these features, CPU manufacturers can offer platforms with a set of security guarantees similar to those provided via dedicated security hardware, such as secure coprocessors, smartcards, or hardware security tokens.

Unfortunately, the nature of these features raises serious challenges for building secure software with guarantees that match those of dedicated trusted hardware. While runtime isolation is important, these features omit many other secure resources present in dedicated trusted hardware, such as storage, secure counters, clocks, and entropy. These omissions raise an important question: *Can we overcome the limitations of commodity CPU se-*

curity features to build software systems with security guarantees similar to those of trusted hardware?

In this work, we answer this question by implementing a software-only Trusted Platform Module (TPM) using ARM TrustZone. We demonstrate that the low-level primitives offered by ARM TrustZone and Intel SGX can be used to build systems with high-level trusted computing semantics. Second, we show that these CPU security features can displace the need for dedicated trusted hardware. Third, we demonstrate that these CPU features can offer backward compatibility, a property often very useful in practice. Google and Microsoft already offer operating systems that leverage commodity TPMs. Building a backwards compatible TPM in software means that no changes are needed to Google and Microsoft operating systems. Finally, we describe a set of mechanisms needed for our software-only TPM that can also be useful for building more sophisticated trusted applications beyond just a TPM.

This paper presents firmware-TPM (fTPM), an end-to-end implementation of a TPM using ARM TrustZone. fTPM provides security guarantees similar, although not identical, to a discrete TPM chip. Our implementation is the reference implementation used in all ARM-based mobile devices running Windows including Microsoft Surface and Windows Phone, comprising millions of mobile devices. fTPM was the first hardware or software implementation to support the newly released TPM 2.0 specification. The fTPM has much better performance than TPM chips and is fully backwards compatible: *no modifications are required to the OS services or applications between a mobile device equipped with a TPM chip and one equipped with an fTPM; all modifications are limited only to firmware and drivers.*

To address the above question, this paper starts with an analysis of ARM TrustZone’s security guarantees. We thoroughly examine the shortcomings of the ARM TrustZone technology needed for building secure services, whether for fTPM or others. We also examine Intel’s

*Currently with ContainerX.

SGX and show that many of TrustZone’s shortcomings remain present.

We present three approaches to overcome the limitations of ARM TrustZone: (1) provisioning additional trusted hardware, (2) making design compromises that do not affect TPM’s security and (3) slightly changing the semantics of a small number of TPM 2.0 commands to adapt them to TrustZone’s limitations. Based on these approaches, our implementation uses a variety of mechanisms, such as *cooperative checkpointing*, *fate sharing*, and *atomic updates*, that help the fTPM overcome the limitations of commodity CPU security features. This paper demonstrates that these limitations can be overcome or compensated for when building a software-only implementation of a dedicated trusted hardware component, such as a TPM chip. The fTPM has been deployed in millions of mobile devices running legacy operating systems and applications originally designed for discrete TPM chips.

Finally, this paper omits some low-level details of our implementation and a more extensive set of performance results. These can be found in the fTPM technical report [44].

2 Trusted Platform Module: An Overview

Although TPMs are more than a decade old, we are seeing a resurgence of interest in TPMs from both industry and the research community. TPMs have had a mixed history, in part due to the initial perception that the primary use for TPMs would be to enable digital rights management (DRM). TPMs were seen as a mechanism to force users to give up control of their own machines to corporations. Another factor was the spotty security record of some the early TPM specifications: TPM version 1.1 [52] was shown to be vulnerable to an unsophisticated attack, known as the *PIN reset* attack [49]. Over time, however, TPMs have been able to overcome their mixed reputation, and are now a mainstream component available in many commodity desktops and laptops.

TPMs provide a small set of primitives that can offer a high degree of security assurance. First, TPMs offer strong machine identities. A TPM can be equipped with a unique RSA key pair whose private key never leaves the physical perimeter of a TPM chip. Such a key can effectively act as a globally unique, unforgeable machine identity. Additionally, TPMs can prevent undesired (i.e., malicious) software rollbacks, can offer isolated and secure storage of credentials on behalf of applications or users, and can attest the identity of the software running on the machine. Both industry and the research community have used these primitives as building blocks in a variety of secure systems. This section presents several such systems.

2.1 TPM-based Secure Systems in Industry

Microsoft. Modern versions of the Windows OS use TPMs to offer features, such as BitLocker disk encryption, virtual smart cards, early launch anti-malware (ELAM), and key and device health attestations.

BitLocker [37] is a full-disk encryption system that uses the TPM to protect the encryption keys. Because the decryption keys are locked by the TPM, an attacker cannot read the data just by removing a hard disk and installing it in another computer. During the startup process, the TPM releases the decryption keys only after comparing a hash of OS configuration values with a snapshot taken earlier. This verifies the integrity of the Windows OS startup process. BitLocker has been offered since 2007 when it was made available in Windows Vista.

Virtual smart cards [38] use the TPM to emulate the functionality of physical smart cards, rather than requiring the use of a separate physical smart card and reader. Virtual smart cards are created in the TPM and offer similar properties to physical smart cards – their keys are not exportable from the TPM, and the cryptography is isolated from the rest of the system.

ELAM [35] enables Windows to launch anti-malware before any third-party drivers or applications. The anti-malware software can be first- or third-party (e.g., Microsoft Windows Defender or Symantec Endpoint Protection). Finally, Windows also uses the TPM to construct attestations of cryptographic keys and device boot parameters [36]. Enterprise IT managers use these attestations to assess the health of devices they manage. A common use is to gate access to high-value network resources based on its attestations.

Google. Modern versions of Chrome OS [22] use TPMs for a variety of tasks, including software and firmware rollback prevention, protecting user data encryption keys, and attesting the mode of a device.

Automatic updates enable a remote party (e.g., Google) to update the firmware or the OS of devices that run Chrome OS. Such devices are vulnerable to “remote rollback attacks”, where a remote attacker replaces newer software, through a difficult-to-exploit vulnerability, with older software, with a well-known and easy-to-exploit vulnerability. Chrome devices use the TPM to prevent software updates to versions older than the current one.

eCryptfs [14] is a disk encryption system used by Chrome OS to protect user data. Chrome OS uses the TPM to rate limit password guessing on the file system encryption key. Any attempt to guess the AES keys requires the use of a TPM, a single-threaded device that

is relatively slow. This prevents parallelized attacks and limits the effectiveness of password brute-force attacks.

Chrome devices can be booted into one of four different modes, corresponding to the state of the developer switch and the recovery switch at power on. These switches may be physically present on the device, or they may be virtual, in which case they are triggered by certain key presses at power on. Chrome OS uses the TPM to attest the device’s current mode to any software running on the machine, a feature used for reporting policy compliance.

More details on the additional ways in which Chrome devices make use of TPMs are described in [22].

2.2 TPM-Based Secure Systems in Research

The research community has proposed many uses for TPMs in recent years.

- **Secure VMs for the cloud:** Software stacks in typical multi-tenant clouds are large and complex, and thus prone to compromise or abuse from adversaries including the cloud operators, which may lead to leakage of security-sensitive data. CloudVisor [58] and Credo [43] are virtualization-approaches that protect the privacy and integrity of customer’s VMs on commodity cloud infrastructure, even when the virtual machine monitor (VMM) or the management VM becomes compromised. These systems require TPMs to attest to cloud customers the secure configuration of the hosts running their VMs.
- **Secure applications, OSs and hypervisors:** Flicker [33], TrustVisor [32], Memoir [41] leverage the TPM to provide various (but limited) forms of runtimes with strong code and data integrity and confidentiality. Code running in these runtimes is protected from the rest of the OS. These systems have small TCBs because they exclude the bulk of the OS.
- **Novel secure functionality:** Pasture [30] is a secure messaging and logging library that provides secure offline data access. Pasture leverages the TPM to provide two safety properties: access-undeniability (a user cannot deny any offline data access obtained by his device without failing an audit) and verifiable-revocation (a user who generates a verifiable proof of revocation of unaccessed data can never access that data in the future). These two properties are essential to an offline video rental service or to an offline logging and revocation service.

Policy-sealed data [47] relies on TPMs to provide a new abstraction for cloud services that lets data be sealed (i.e., encrypted to a customer-defined policy) and then unsealed (i.e., decrypted) only by hosts whose configu-

rations match the policy.

cTPM [9] extends the TPM functionality across several devices as long as they are owned by the same user. cTPM thus offers strong user identities (across all of her devices), and cross-device isolated secure storage.

Finally, mobile devices can leverage a TPM to offer new trusted services [19, 31, 28]. One example is *trusted sensors* whose readings have a high degree of authenticity and integrity. Trusted sensors enable new mobile apps relevant to scenarios in which sensor readings are very valuable, such as finance (e.g., cash transfers and deposits) and health (e.g., gather health data) [48, 56]. Another example is enforcing driver distraction regulations for in-car music or navigation systems [28].

2.3 TPM 2.0: A New TPM Specification

The Trusted Computing Group (TCG) recently defined the specification for TPM version 2.0 [54]. This newer TPM is needed for two key reasons. First, the crypto algorithms in TPM 1.2 [55] have become inadequate. For example, TPM 1.2 only offers SHA-1 and not SHA-2; SHA-1 is now considered weak and cryptographers are reluctant to use it. Another example is the introduction of ECC with TPM 2.0.

The second reason is the lack of an universally-accepted reference implementation of the TPM 1.2 specification. As a result, different TPM 1.2 implementations exhibit slightly different behaviors. The lack of a reference implementation also keeps the TPM 1.2 specification ambiguous. It can be difficult to specify the exact behavior of cryptographic protocols in English. Instead, with TPM 2.0 the specification is the same as the reference implementation. The specification consists of several documents describing the behavior of the codebase, and these documents are derived directly from the TPM 2.0 codebase, thereby ensuring uniform behavior.

Recently, TPM manufacturers have started to release discrete chips implementing TPM 2.0. Also, at least one manufacturer has released a firmware upgrade that can update a TPM 1.2 chip into one that implements both TPM 2.0 and TPM 1.2. Note that although TPM 2.0 subsumes the functionality of TPM 1.2, it is not backwards compatible. A BIOS built to use a TPM 1.2 would not work with a TPM 2.0-only chip. A list of differences between the two versions is provided by the TCG [53].

3 Modern Trusted Computing Hardware

Recognizing the increasing demand for security, modern CPUs have started to incorporate trusted computing features, such as ARM TrustZone [1] and Intel Software Guard Extensions (SGX) [25]. This section presents

the background on ARM TrustZone (including its shortcomings); this background is important to the design of fTPM. Later, Section 12 will describe Intel’s SGX and its shortcomings.

3.1 ARM TrustZone

ARM TrustZone is ARM’s hardware support for trusted computing. It is a set of security extensions found in many recent ARM processors (including Cortex A8, Cortex A9, and Cortex A15). ARM TrustZone provides two virtual processors backed by hardware access control. The software stack can switch between the two states, referred to as “worlds”. One world is called *secure world* (SW), and the other *normal world* (NW). Each world acts as a runtime environment with its own resources (e.g., memory, processor, cache, controllers, interrupts). Depending on the specifics of an individual ARM SoC, a single resource can be strongly partitioned between the two worlds, can be shared across worlds, or assigned to a single world only. For example, most ARM SoCs offer memory curtaining, where a region of memory can be dedicated to the secure world. Similarly, processor, caches, and controllers are often shared across worlds. Finally, I/O devices can be mapped to only one world, although on certain SoCs this mapping can be dynamically controlled by a trusted peripheral.

- **Secure monitor:** The secure monitor is an ARM processor mode that enables context switching between the secure and normal worlds. A special register determines whether the processor core runs code in the secure or non-secure worlds. When the core runs in monitor mode the processor is considered secure regardless of the value of this register.

An ARM CPU has separate banks of registers for each of the two worlds. Each of the worlds can only access their separate register files; cross-world register access is blocked. However, the secure monitor can access non-secure banked copies of registers. The monitor can thus implement context switches between the two worlds.

- **Secure world entry/exit:** By design, an ARM platform always boots into the secure world first. Here, the system firmware can provision the runtime environment of the secure world before any untrusted code (e.g., the OS) has a chance to run. For example, the firmware allocates secure memory for TrustZone, programs the DMA controllers to be TrustZone-aware, and initializes any secure code. The secure code eventually yields to the normal world where untrusted code can start executing.

The normal world uses a special ARM instruction called *smc* (secure monitor call) to transfer control into the secure world. When the CPU executes the *smc* instruction, the hardware switches into the secure monitor,

which performs a secure context switch into the secure world. Hardware interrupts can trap directly into the secure monitor code, which enables flexible routing of those interrupts to either world. This allows I/O devices to map their interrupts to the secure world if desired.

- **Curtained memory:** At boot time, the software running in the secure monitor can allocate a range of physical addresses to the secure world only, creating the abstraction of curtained memory – memory inaccessible to the rest of the system. For this, ARM adds an extra control signal for each of the read and write channels on the main memory bus. This signal corresponds to an extra bit (a 33rd-bit on a 32-bit architecture) called the *non-secure* bit (NS-bit). These bits are interpreted whenever a memory access occurs. If the NS-bit is set, an access to memory allocated to the secure world fails.

3.2 Shortcomings of ARM TrustZone

Although the ARM TrustZone specification describes how the processor and memory subsystem are protected in the secure world and provides mechanisms for securing I/O devices, the specification is silent on how many other resources should be protected. This has led to fragmentation – SoCs offer various forms of protecting different hardware resources for TrustZone, or no protection at all. While there may be major differences between the ARM SoCs offered by different vendors, the observations below held across all the major SoCs vendors when products based on this work shipped.

- **No Trusted Storage:** Surprisingly, the ARM TrustZone specification offers no guidelines on how to implement secure storage for TrustZone. The lack of secure storage drastically reduces the effectiveness of TrustZone as trusted computing hardware.

Naively, one might think that code in TrustZone could encrypt its persistent state and store it on untrusted storage. However, encryption alone is not sufficient because (1) one needs a way to store the encryption keys securely, and (2) encryption cannot prevent rollback attacks.

- **Lack of Secure Entropy and Persistent Counters:** Most trusted systems make use of cryptography. However, the TrustZone specification is silent on offering a secure entropy source or a monotonically increasing persistent counter. As a result, most SoCs lack an entropy pool that can only be read from the secure world, and a counter that can persist across reboots and cannot be incremented by the normal world.

- **Lack of virtualization:** Sharing the processor across two different worlds in a stable manner can be done using virtualization techniques. Although ARM offers virtualization extensions [2], the ARM TrustZone specification

does not mandate them. As a result, many ARM-based SoCs used in mobile devices today lack virtualization support. Virtualizing commodity operating systems on an ARM platform lacking hardware-assistance for virtualization is challenging.

- **Lack of secure clock and other peripherals:** Secure systems often require a secure clock. While TrustZone can protect memory, interrupts, and certain system buses on the SoC, this protection does not extend to the ARM peripheral bus. It is hard to reason about the security guarantees of a peripheral if its controller can be programmed by the normal world, even when its interrupts and memory region are mapped into the secure world. Malicious code could program the peripheral in a way that could make it insecure. For example, some peripherals could be put in “debug mode” to generate arbitrary readings that do not correspond to the ground truth.
- **Lack of access:** Most SoC hardware vendors do not provide access to their firmware. As a result, many developers and researchers are unable to find ways to deploy their systems or prototypes to TrustZone. In our experience, this has seriously impeded the adoption of TrustZone as a trusted computing mechanism.

SoC vendors are reluctant to give access to their firmware. They argue that their platforms should be “locked down” to reduce the likelihood of “hard-to-remove” rootkits. Informally, SoC vendors also perceive firmware access as a threat to their competitiveness. They often incorporate proprietary algorithms and code into their firmware that takes advantage of the vendor-specific features offered by the SoC. Opening firmware to third parties could expose more details about these features to their competitors.

4 High-Level Architecture

Leveraging ARM TrustZone, we implement a trusted execution environment (TEE) that acts as a basic operating system for the secure world. Figure 1 illustrates our architecture, and our system’s trusted computing base (TCB) is shown in the shaded boxes.

At a high-level, the TEE consists of a monitor, a dispatcher, and a runtime where one or more trusted services (such as the fTPM) can run one at a time. The TEE exposes a single trusted service interface to the normal world using shared memory. Our system’s TCB comprises the ARM SoC hardware, the TEE layers, and the fTPM service.

By leveraging the isolation properties of ARM TrustZone, the TEE provides *shielded execution*, a term coined by previous work [5]. With shielded execution, the TEE offers two security guarantees:

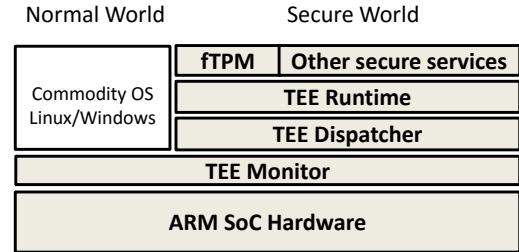


Figure 1: **The architecture of the fTPM.** This diagram is not to scale.

- *Confidentiality:* The whole execution of the fTPM (including its secrets and execution state) is hidden from the rest of the system. Only the fTPM’s inputs and outputs, but no intermediate states, are observable.
- *Integrity:* The operating system cannot affect the behavior of the fTPM, except by choosing to refuse execution or to prevent access to system’s resources (DoS attacks). The fTPM’s commands are always executed correctly according to the TPM 2.0 specification.

4.1 Threat Model and Assumptions

A primary assumption is that the commodity OS running in the Normal World is untrusted and potentially compromised. This OS could mount various attacks to code running in TrustZone, such as making invalid calls to TrustZone (or setting invalid parameters), not responding to requests coming from TrustZone, or responding incorrectly. In handling these attacks, it is important to distinguish between two cases: (1) not handling or answering TrustZone’s requests, or (2) acting maliciously.

The first class of attacks corresponds to *refusing service*, a form of Denial-of-Service attacks. DoS attacks are out of scope according to the TPM 2.0 specification. These attacks cannot be prevented as long as an untrusted commodity OS has access to platform resources, such as storage or network. For example, a compromised OS could mount various DoS attacks, such as erasing all storage, resetting the network card, or refusing to call the *smc* instruction. Although our fTPM will remain secure (e.g., preserves confidentiality and integrity of its data) in the face of these attacks, the malicious OS could starve the fTPM leaving it inaccessible.

However, the fTPM must behave correctly when the untrusted OS makes incorrect requests, returns unusual values (or fails to return at all), corrupts data stored on stable storage, injects spurious exceptions, or sets the platform clock to an arbitrary value.

At the hardware level, we assume that the ARM SoC (including ARM TrustZone) itself is implemented correctly, and is not compromised. An attacker cannot

mount hardware attacks to inspect the contents of the ARM SoC, nor the contents of RAM memory on the platform. However, the adversary has full control beyond the physical boundaries of the processor and memory. They may read the flash storage and arbitrarily alter I/O including network traffic or any sensors found on the mobile device. In other work, we address the issue of physical attacks on the memory of a mobile device [10].

We defend against side-channel attacks that can be mounted by malicious software. Cache collision attacks are prevented because all caches are flushed when the processor context switches to and from the Secure World. Our fTPM implementation’s cryptography library uses constant time cryptography and several other timing attack preventions, such as RSA blinding [27]. However, we do not defend against power analysis or other side-channel attacks that require physical access to hardware or hardware modifications.

We turn our focus on the approaches taken to overcome TrustZone’s shortcomings in the fTPM.

5 Overcoming TrustZone Shortcomings

We used three approaches to overcome the shortcomings of ARM TrustZone’s technology.

- **Approach #1: Hardware Requirements.** Providing secure storage to TEE was a serious concern. One option was to store the TEE’s secure state in the cloud. We dismissed this alternative as not viable because of its drastic impact on device usability. TPMs are used to measure the boot software (including the firmware) on a device. A mobile device would then require cloud connectivity at boot time in order to download the fTPM’s state and start measuring the boot software.

Instead, we imposed additional hardware requirements on device manufacturers to ensure a minimum level of hardware support for the fTPM. Many mobile devices already come equipped with an embedded Multi-Media Controller (eMMC) storage controller that has an (off-SoC) replay-protected memory block (RPMB). The RPMB’s presence, combined with encryption, ensures that TEE can offer storage that meets the needs of all the fTPM’s security properties. Thus, our first hardware requirement for TEE is an eMMC controller with support for RPMB.

Second, we require the presence of hardware fuses accessible only from the secure world. A hardware fuse provides write-once storage. At provisioning time (before being released to a retail store), manufacturers provision our mobile devices by setting the secure hardware fuses with a secure key unique per device. We also require an entropy source accessible from the secure world.

The TEE uses both the secure key and the entropy source to generate cryptographic keys at boot time.

Section 6 provides in-depth details of these three hardware requirements.

- **Approach #2: Design Compromises.** Another big concern was long-running TEE commands. Running inside TrustZone for a long time could jeopardize the stability of the commodity OS. Generally, sharing the processor across two different worlds in a stable manner should be done using virtualization techniques. Unfortunately, many of the targeted ARM platforms lack virtualization support. Speaking to the hardware vendors, we learned that it is unlikely virtualization will be added to their platforms any time soon.

Instead, we compromised and require that no TEE code path can execute for long periods of time. This translates into an fTPM requirement – no TPM 2.0 command can be long running. Our measurements of TPM commands revealed that only one TPM 2.0 command is long running: generating RSA keys. Section 7 presents the compromise made in the fTPM design when an RSA key generation command is issued.

- **Approach #3: Modifying the TPM 2.0 Semantics.** Lastly, we do not require the presence of a secure clock from the hardware vendors. Instead, the platform only has a secure timer that ticks at a pre-determined rate. We thus determined that the fTPM cannot offer any TPM commands that require a clock for their security. Fortunately, we discovered that some (but not all) TPM commands can still be offered by relying on a secure timer albeit with slightly altered semantics. Section 8 will describe all these changes in more depth.

6 Hardware Requirements

6.1 eMMC with RPMB

eMMC stands for embedded Multi-Media Controller, and refers to a package consisting of both flash memory and a flash memory controller integrated on the same silicon die [11]. eMMC consists of the MMC (multimedia card) interface, the flash memory, and the flash memory controller. Later versions of the eMMC standard offer a replay-protected memory block (RPMB) partition. As the name suggests, RPMB is a mechanism for storing data in an authenticated and replay-protected manner.

RPMB’s replay protection utilizes three mechanisms: an authentication key, a write counter, and a nonce.

RPMB Authentication Key: A 32-byte one-time programmable authentication key register. Once written, this register cannot be over-written, erased, or even read. The eMMC controller uses this authentication key to compute HMACs (SHA-256) to protect data integrity.

Programming the RPMB authentication key is done by issuing a specially formatted dataframe. Next, a result read request dataframe must be also issued to check that the programming step succeeded. Access to the RPMB is prevented unless the authentication key has been programmed. Any write/read requests will return a special error code indicating that the authentication key has yet to be programmed.

RPMB Write Counter: The RPMB partition also maintains a counter for the number of authenticated write requests made to RPMB. This is a 32-bit counter initially set to 0. Once it reaches its maximum value, the counter will no longer be incremented and a special bit will be turned on in all dataframes to indicate that the write counter has expired. The correct counter value must be included in each dataframe written to the controller.

Nonce: RPMB allows a caller to label its read requests with 16-byte nonces that are reflected in the read responses. These nonces ensure that reads are fresh.

6.1.1 Protection against replay attacks

To protect writes from replay attacks, each write includes a write counter value whose integrity is protected by an authentication key (the RPMB authentication key), a shared secret provisioned into both the secure world and the eMMC controller. The read request dataframe that verifies a write operation returns the incremented counter value, whose integrity is protected by the RPMB authentication key. This ensures that the write request has been successful.

The role of nonces in read operations protects them against replay attacks. To ensure freshness, whenever a read operation is issued, the request includes a nonce and the read response includes the nonce signed with RPMB authentication key.

6.2 Secure World Hardware Fuses

We required a set of hardware fuses that can be read from the secure world only. These fuses are provisioned with a hard-to-guess, unique-per-device number. This number is used as a seed in deriving additional secret keys used by the fTPM. Section 9 will describe in-depth how the seed is used in deriving secret fTPM keys, such as the secure storage key (SSK).

6.3 Secure Entropy Source

The TPM specification requires a true random number generator (RNG). A true RNG is constructed by having an entropy pool whose entropy is supplied by a hardware

oscillator. The secure world must manage this pool because the TEE must read from it periodically.

Generating entropy is often done via some physical process (e.g., a noise generator). Furthermore, an entropy generator has a *rate of entropy* that specifies how many bits of entropy are generated per second. When the platform is first started, it can take some time until it has gathered “enough” bits of entropy for a seed.

We require the platform manufacturer to provision an entropy source that has two properties: (1) it can be managed by the secure world, and (2) its specification lists a conservative bound on its rate of entropy; this bound is provided as a configuration variable to the fTPM. Upon a platform start, the fTPM waits to initialize until sufficient bits of entropy are generated. For example, the fTPM would need to wait at least 25 seconds to initialize if it requires 500 bits of true entropy bits from a source whose a rate is 20 bits/second.

Alerted to this issue, the TPM 2.0 specification has added the ability to save and restore any accumulated but unused entropy across reboots. This can help the fTPM reduce the wait time for accumulating entropy.

7 Design Compromises

7.1 Background on Creating RSA Keys

Creating an RSA key is a resource-intensive operation for two reasons. First, it requires searching for two large prime numbers, and such a search is theoretically unbounded. Although many optimizations exist on how to search RSA keys efficiently [40], searching for keys is still a lengthy operation. Second, the search must be seeded with a random number, otherwise an attacker could attempt to guess the primes the search produced. Thus the TPM cannot create an RSA key unless the entropy source has produced enough entropy to seed the search.

The TPM can be initialized with a *primary* storage root key (SRK). The SRK’s private portion never leaves the TPM and is used in many TPM commands (such as TPM *seal* and *unseal*). Upon TPM initialization, our fTPM waits to accumulate the entropy required to seed the search for large prime numbers. The fTPM also creates RSA keys upon receiving a *create RSA keys* command¹.

TPM 2.0 checks whether a number is prime using the Miller-Rabin probabilistic primality test [40]. If the test fails, the candidate number is not a prime. However, upon passing, the test offers a probabilistic guarantee – the candidate is likely a prime with high probability. The TPM repeats this test a couple of times to increase

¹This corresponds to the TPM 2.0 TPM2>Create command.

the likelihood the candidate is prime. Choosing a composite number during RSA key creation has catastrophic security consequences because it allows an attacker to recover secrets protected by that key. TPM 2.0 repeats the primality test five times for RSA-1024 keys and four times for all RSA versions with longer keys. This reduces the likelihood of choosing a false prime to a probability lower than 2^{-100} .

7.2 Cooperative Checkpointing

Our fTPM targets several different ARM platforms (from smartphones to tablets) that lack virtualization support, and the minimal OS in our TEE lacks a preemptive scheduler. Therefore, we impose a requirement on services running in the TEE that the transitions to TEE and back must be short to ensure that the commodity OS remains stable. Unfortunately, creating an RSA key is a very long process, often taking in excess of 10 seconds on our early hardware tablets.

Faced with this challenge, we added *cooperative checkpointing* to the fTPM. Whenever a TPM command takes too long, the fTPM checkpoints its state in memory, and returns a special error code to the commodity OS running in the Normal World.

Once the OS resumes running in the Normal World, the OS is free to call back the TPM command and instruct the fTPM to resume its execution. These “resume” commands continue processing until the command completes or the next checkpoint occurs. Additionally, the fTPM also allows all commands to be cancelled. The commodity OS can cancel any TPM command even when in the command is in a checkpointed state. Cooperative checkpointing lets us bypass the lack of virtualization support in ARM, yet continue to offer long-running TPM commands, such as creating RSA keys.

8 Modifying TPM 2.0 Semantics

8.1 Secure Clock

TPMs use secure clocks for two reasons. The first use is to measure lockout durations. Lockouts are time periods when the TPM refuses service. Lockouts are very important to authorizations (e.g., checking a password). If a password is incorrectly entered more than k times (for a small k), the TPM enters lockout and refuses service for a pre-determined period of time. This thwarts dictionary attacks – guessing a password incorrectly more than k times puts the TPM in lockout mode.

The second use of a secure clock in TPMs is for time-bound authorizations, such as the issuing an authorization valid for a pre-specified period of time. For example, the TPM can create a key valid for an hour only. At

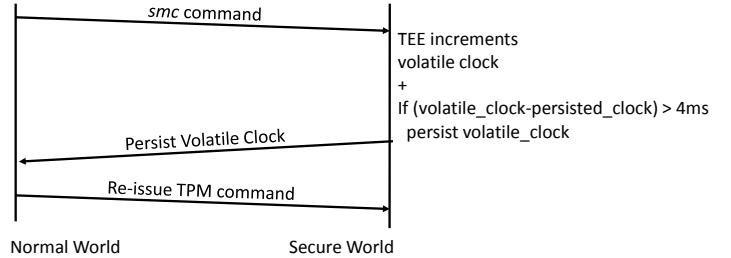


Figure 2: **fTPM clock update.**

the end of an hour, the key becomes unusable.

8.1.1 Requirements of the TPM 2.0 Specification

A TPM 2.0 requirement is the presence of a clock with millisecond granularity. The TPM uses this clock only to measure intervals of time for time-bound authorizations and lockouts. The volatile clock value must be persisted periodically to a specially-designed non-volatile entry called *NVClock*. The periodicity of the persistence is a TPM configuration variable and cannot be longer than 2^{22} milliseconds (~70 minutes).

The combination of these properties ensures that the TPM clock offers the following two guarantees: 1. *the clock advances while the TPM is powered*, 2. *the clock never rolls backwards more than NVClock update periodicity*. The only time when the clock can roll backward is when the TPM loses power right before persisting the NVClock value. Upon restoring power, the clock will be restored from NVClock and thus rolled back. The TPM also provides a flag that indicates the clock may have been rolled back. This flag is cleared when the TPM can guarantee the current clock value could not have been rolled back.

Given these guarantees, the TPM can measure time only while the platform is powered up. For example, the TPM can measure one hour of time as long as the platform does not reboot or shutdown. However, the clock can advance slower than wall clock *but only due to a reboot*. Even in this case time-bound authorizations are secure because they do not survive reboots by construction: in TPM 2.0, a platform reboot automatically expires all time-bound authorizations.

8.1.2 Fate Sharing

The main difficulty in building a secure clock in the fTPM is that persisting the clock to storage requires the cooperation of the (untrusted) OS. The OS could refuse to perform any writes that would update the clock. This would make it possible to roll back the clock arbitrarily just by simply rebooting the platform.

The fate sharing model suggests that it is acceptable to lose the clock semantics of the TPM as long as the TPM itself becomes unusable. Armed with this principle, we designed the fTPM’s secure clock to be the first piece of functionality the fTPM executes on *all* commands. The fTPM refuses to provide any functionality until the clock is persisted. Figure 2 illustrates how the fTPM updates its clock when the TEE is scheduled to run.

The fTPM implementation does not guarantee that the clock cannot be rolled back arbitrarily. For example, an OS can always refuse to persist the fTPM’s clock for a long time, and then reboot the platform effectively rolling back the clock. However, fate sharing guarantees that the fTPM services commands *if and only if* the clock behaves according to the TPM specification.

8.2 Dark Periods

The diversity of mobile device manufacturers raised an additional challenge for implementing the fTPM. A mobile device boot cycle starts by running firmware developed by one (of the many) hardware manufacturers, and then boots a commodity OS. The fTPM must provide functionality throughout the entire boot cycle. In particular, both Chrome and Windows devices issue TPM *Unseal* commands after the firmware finishes running, but before the OS starts booting. These commands attempt to unseal the decryption keys required for decrypting the OS loader. At this point, the fTPM cannot rely on external secure storage because the firmware has unloaded its storage drivers while the OS has yet to load its own. We refer to this point as a “dark period”.

TPM Unseal uses storage to record a failed unseal attempt. After a small number of failed attempts, the TPM enters lockout and refuses service for a period of time. This mechanism rate-limits the number of attempts to guess the unseal authorization (e.g., Windows lets users enter a PIN number to unseal the OS loader using BitLocker). The TPM maintains a counter of failed attempts and requires persisting it each time the counter increments. This eliminates the possibility of an attacker brute-forcing the unseal authorization and rebooting the platform without persisting the counter. Figures 3, 4, and 5 illustrate three timelines: a TPM storing its failed attempts counter to stable storage, a TPM without stable storage being attacked with by a simple reboot, and the fTPM solution to dark periods based on the dirty bit.

8.2.1 Modifying the Semantics of Failed Tries

We address the lack of storage during a dark period by making a slight change in how the TPM 2.0 interprets the failed tries counter. At platform boot time, before entering any possible dark periods, the fTPM persists

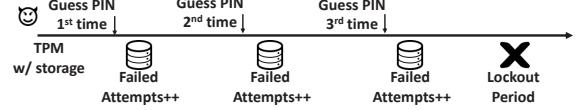


Figure 3: **TPM with storage (no dark period).** TPM enters lockout if adversary makes too many guess attempts. This sequence of steps is secure.

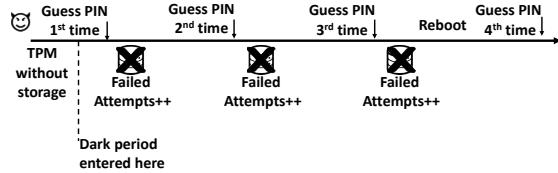


Figure 4: **TPM during a dark period (no stable storage).** Without storing the failed attempts counter, the adversary can simply reboot and avoid TPM lockout. This sequence of steps is insecure.

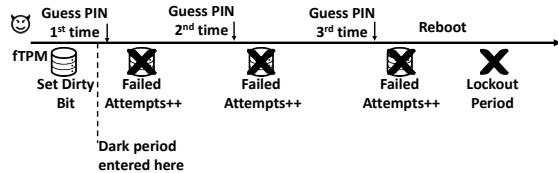


Figure 5: **fTPM during a dark period (no stable storage).** fTPM sets the dirty bit before entering a dark period. If reboot occurs during the dark period, fTPM enters lockout automatically. This sequence of steps is secure.

a *dirty bit*. If for any reason the fTPM is unable to persist the dirty bit, it refuses to offer service. If the dark period is entered and the unseal succeeds, the OS will start booting successfully and load its storage drivers. Once storage becomes available again, the dirty bit is cleared. However, the dirty bit remains uncleared should the mobile device reboot during a dark period. In this case, when the fTPM initializes and sees that the bit is dirty, the fTPM cannot distinguish between a legitimate device reboot (during a dark period) and an attack attempting to rollback the failed tries counter. Conservatively, the fTPM assumes it is under attack, the counter is immediately incremented to the maximum number of failed attempts, and the TPM enters lockout.

This change in semantics guarantees that an attack against the counter remains ineffective. The trade-off is that a legitimate device reboot during a dark period puts the TPM in lockout. The TPM cannot unseal until the lockout duration expires (typically several minutes).

Alerted to this problem, the TPM 2.0 designers have added a form of the *dirty bit* to their specification, called

the *non-orderly* or *unordered bit* (both terms appear in the specification). Unfortunately, they did not adopt the idea of having a small number of tries before the TPM enters lockout mode. Instead, the specification dictates that the TPM enters lockout as soon as a failed unsealed attempt cannot be recorded to storage. Such a solution impacts usability because it locks the TPM as soon as the user has entered an incorrect PIN or password.

9 Providing Storage to Secure Services

The combination of encryption, the RPMB, and hardware fuses is sufficient to build trusted storage for the TEE. Upon booting the first time, TEE generates a symmetric RPMB key and programs it into the RPMB controller. The RPMB key is derived from existing keys available on the platform. In particular, we construct a secure storage key (SSK) that is unique to the device and derived as following:

$$SSK := KDF(HF, DK, UUID) \quad (1)$$

where KDF is a one-way key derivation function, HF is the value read from the hardware fuses, DK is a device key available to both secure and normal worlds, and UUID is the device’s unique identifier.

The SSK is used for authenticated reads and writes of all TEE’s persistent state (including the fTPM’s state) to the device’s flash memory. Before being persisted, the state is encrypted with a key available to TrustZone only. Encryption ensures that all fTPM’s state remains confidential and integrity protected. The RPMB’s authenticated reads and writes ensure that fTPM’s state is also resilient against replay attacks.

9.1 Atomic Updates

TEE implements atomic updates to the RPMB partition. Atomic updates are necessary for fTPM commands that require multiple separate write operations. If these writes are not executed atomically, TEE’s persistent state could become inconsistent upon a failure that leaves the secure world unable to read its state.

The persisted state of the fTPM consists of a sequence of blocks. TEE stores two copies of each block: one representing the committed version of the state block and one its shadow (or uncommitted) version. Each block id X has a corresponding block whose id is $X + N$, where N is the size of fTPM’s state. The TEE also stores a bit vector in its first RPMB block. Each bit in this vector indicates which block is committed: if the i bit is 0 then the i th block committed id is X , otherwise is $X + N$. In this way, all pending writes to shadow blocks are committed using a single atomic write operation of the bit vector.



Figure 6: **RMPB blocks.** Bit vector mechanism used for atomic updates.

Allocating the first RPMB entry to the bit vector limits the size of the RPMB partition to 256KB (the current eMMC specification limits the size of a block to 256 bytes). If that size is insufficient, an extra layer of indirection can extend the bit vector mechanism to support up to 512MB ($256 * 8 * 256 * 8 = 1,048,576$ blocks).

Figure 6 illustrates the bit vector mechanism for atomic updates. On the left, the bit vector shows which block is committed (bit value 0) and which block is shadow (bit value 1). The committed blocks are shown in solid color.

In the future, we plan to improve the fTPM’s performance by offering transactions to fTPM commands. All writes in a transaction are cached in memory and persisted only upon commit. The commit operation first updates the shadow version of changed blocks, and then updates the metadata in a single atomic operation to make shadow version for updated blocks the committed version. A command that updates secure state must either call commit or abort before returning. Abort is called implicitly if commit fails, where shadow copy is rolled back to the last committed version, and an error code is returned. In this scenario, the command must implement rollback of any in-memory data structure by itself.

10 Performance Evaluation

This paper answers two important questions on performance²:

1. What is the overhead of long-running fTPM commands such as *create RSA keys*? The goal is to shed light on the fTPM implementation’s performance when seeking prime numbers for RSA keys.

2. What is the performance overhead of typical fTPM commands, and how does it compare to the performance of a discrete TPM chip? TPM chips have notoriously slow microcontrollers [33]. In contrast, fTPM commands execute on full-fledged ARM cores.

10.1 Methodology

To answer these questions, we instrumented four off-the-shelf commodity mobile devices equipped with fTPMs and three machines equipped with discrete TPMs. We keep these devices’ identities confidential, and refer to

²The fTPM technical report presents additional results of the performance evaluation [44].

fTPM Device	Processor Type
Device # fTPM ₁	1.2 GHz Cortex-A7
Device # fTPM ₂	1.3 GHz Cortex-A9
Device # fTPM ₃	2 GHz Cortex-A57
Device # fTPM ₄	2.2 GHz Cortex-A57

Table 1: Description of fTPM-equipped devices used the evaluation.

them as fTPM₁ through fTPM₄, and dTPM₁ through dTPM₃. All mobile devices are commercially available both in USA and the rest of the world and can be found in the shops of most cellular carriers. Similarly, the discrete TPM 2.0 chips are commercially available. Table 1 describes the characteristics of the mobile ARM SoC processors present in the fTPM-equipped devices. The only modifications made to these devices’ software is a form of device unlock that lets us load our own test harness and gather the measurement results. These modifications do not interfere with the performance of the fTPM running on the tablet.

Details of TPM 2.0 Commands. To answer the questions raised by our performance evaluation, we created a benchmark suite in which we perform various TPM commands and measure their duration. We were able to use timers with sub-millisecond granularity for all our measurements, except for device fTPM₂. Unfortunately, device fTPM₂ only exposes a timer with a 15-ms granularity to our benchmark suite, and we were not able to unlock its firmware to bypass this limitation.

Each benchmark test was run ten times in a row. Although this section presents a series of graphs that answer our performance evaluation questions, an interested reader can find all the data gathered in our benchmarks in the fTPM technical report [44].

- **Create RSA keys:** This TPM command creates an RSA key pair. When this command is issued, a TPM searches for prime numbers, creates the private and public key portions, encrypts the private portion with a root key, and returns both portions to the caller. We used 2048-bit RSA keys in all our experiments. We chose 2048-bit keys because they are the smallest key size still considered secure (1024-bit keys are considered insecure and their use has been deprecated in most systems).

- **Seal and unseal:** The TPM Seal command takes in a byte array, attaches a policy (such as a set of Platform Configuration Register (PCR) values), encrypts with its own storage key, and returns it to the caller. The TPM Unseal command takes in an encrypted blob, checks the policy, and decrypts the blob if the policy is satisfied by the TPM state (e.g., the PCR values are the same as at seal time). We used a ten-byte input array to Seal, and we set an empty policy.

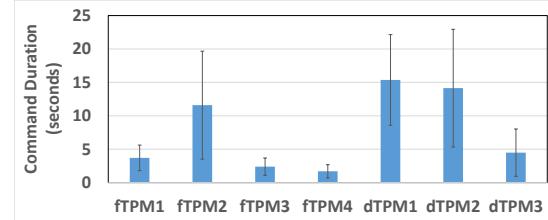


Figure 7: Latency of create RSA-2048 keys on various fTPM and dTPM platforms.

- **Sign and verify:** These TPM commands correspond to RSA sign and verify. We used a 2048-bit RSA key for RSA operations and SHA-256 for integrity protection.
- **Encryption and decryption:** These TPM commands correspond to RSA encryption and decryption. We used a 2048-bit RSA key for RSA operations, OAEP for padding, and SHA-256 for integrity protection.
- **Load:** This TPM command loads a previously-created RSA key into the TPM. This allows subsequent command, such as signing and encryption, to use the preloaded key. We used a 2048-bit RSA key in our TPM Load experiments.

10.2 Overhead of RSA Keys Creation

Figure 7 shows the latency of a TPM create RSA-2048 keys command across all our seven devices. As expected, creating RSA keys is a lengthy command taking several seconds on all platforms. These long latencies justify our choice of using cooperative checkpointing (see Section 7) in the design of the fTPM to avoid leaving the OS suspended for several seconds at a time.

Second, the performance of creating keys can be quite different across devices. fTPM₂ takes a much longer time than all other devices equipped with an fTPM. This is primarily due to the variations in the firmware performance across these devices – some manufacturers spend more time optimizing the firmware running on their platforms than others. Even more surprisingly, the discrete TPM 2.0 chips also have very different performance characteristics: dTPM₃ is much faster than dTPM₁ and dTPM₂. Looking at the raw data (shown in [44]), we believe that dTPM₃ searches for prime numbers in the background, even when no TPM command is issued, and maintains a cache of prime numbers.

Figure 7 also shows that the latency of creating keys has high variability due to how quickly prime numbers are found. To shed more light into the variability of finding prime numbers, we instrumented the fTPM codebase to count the number of prime candidates considered when creating an RSA 2048 key pair. For each test, all candidates are composite numbers (and thus discarded)

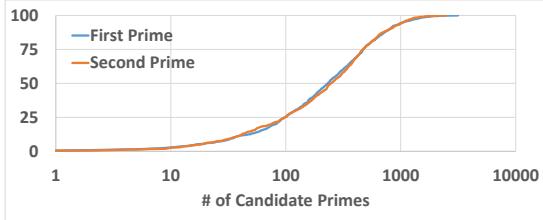


Figure 8: Performance of searching for primes.

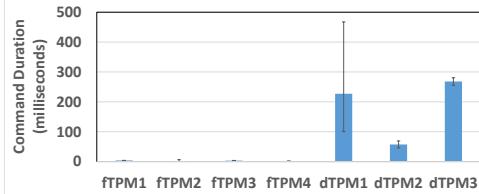


Figure 9: Performance of TPM seal command.



Figure 10: Performance of TPM unseal command.

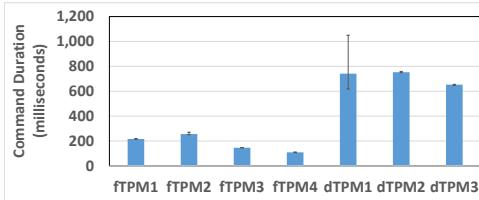


Figure 11: Performance of TPM sign command.

except for the last number. We repeated this test 1,000 times. We plot the cumulative distribution function of the number of candidates for each of the two primes (p and q) in Figure 8. These results demonstrate the large variability in the number of candidate primes considered. While, on average, it takes about 200 candidates until a prime is found (the median was 232 and 247 candidates for p and q , respectively), sometimes a single prime search considers and discards thousands of candidates (the worst case was 3,145 and 2,471 for p and q , respectively).

10.3 Comparing fTPMs to dTPMs

Figures 9–15 show the latencies of several common TPM 2.0 commands. The main result is that fTPMs are much faster than their discrete counterparts. On average, the slowest fTPM is anywhere between 2.4X (for decrypt

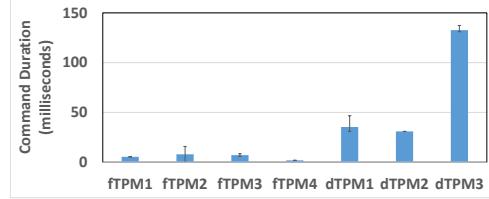


Figure 12: Performance of TPM verify command.

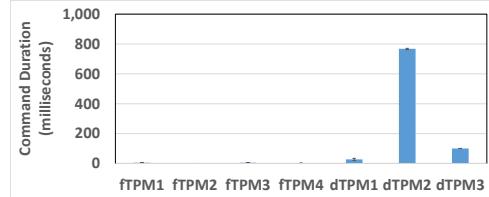


Figure 13: Performance of TPM encrypt command.

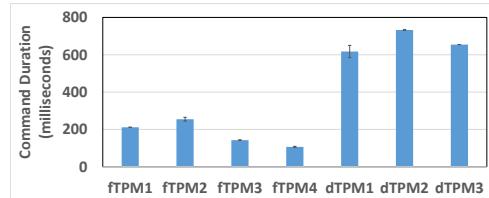


Figure 14: Performance of TPM decrypt command.

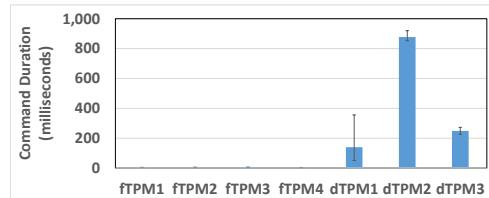


Figure 15: Performance of TPM load command.

tion) and 15.12X (for seal) faster than the fastest dTPM. This is not surprising because fTPMs run their code on ARM Cortex processors, whereas discrete chips are relegated to using much slower microprocessors. The fTPM technical report illustrates these vast performance improvements in even greater detail [44].

These performance results are encouraging. Traditionally, TPMs have not been used for bulk data cryptographic operations due to their performance limitations. With firmware TPMs however, the performance of these operations is limited only by processor speed and memory bandwidth. Furthermore, fTPMs could become even faster by taking advantage of crypto accelerators. Over time, we anticipate that crypto operations will increasingly abandon the OS crypto libraries in favor of the fTPM. This provides increased security as private keys never have to leave TrustZone’s secure perimeter.

10.4 Evaluation Summary

In summary, our evaluation shows that (1) the firmware TPM has better performance than discrete TPM chips, and (2) creating RSA keys is a lengthy operation with high performance variability.

11 Security Analysis

The fTPM’s security guarantees are not identical to those of a discrete TPM chip. This section examines these differences in greater depth.

On- versus off-chip. Discrete TPM chips connect to the CPU via a serial bus; this bus represents a new attack surface because it is externally exposed to an attacker with physical access to the main board. Early TPM chips were attached to the I²C bus, one of the slower CPU buses, that made it possible for an attacker to intercept and issue TPM commands [49]. Modern TPM specifications have instructed the hardware manufacturers to attach the TPM chip to a fast CPU bus and to provide a secure *platform reboot signal*. This signal must guarantee that the TPM reboots (e.g., resets its volatile registers) *if and only if* the platform reboots.

In contrast, by running in the device’s firmware, the fTPM sidesteps this attack surface. The fTPM has no separate bus to the CPU. The fTPM reads its state from secure storage upon initialization, and stores all its state in the CPU and the hardware-protected DRAM.

Memory attacks. By storing its secrets in DRAM, the fTPM is vulnerable to a new class of physical attacks – memory attacks that attempt to read secrets from DRAM. There are different avenues to mount memory attacks, such as cold boot attacks [23, 39], attaching a bus monitor to monitor data transfers between the CPU and system RAM [21, 17, 18], or mounting DMA attacks [6, 8, 42].

In contrast, discrete TPM chips do not make use of the system’s DRAM and are thus resilient to such attacks. However, there is a corresponding attack that attempts to remove the chip’s physical encasing, expose its internal dies, and thus read its secrets. Previous research has already demonstrated the viability of such attacks (typically referred to as *decapping* the TPM), although they remain quite expensive to mount in practice [26].

The fTPM’s susceptibility to memory attacks has led us to investigate inexpensive counter-measures. Sentry is a prototype that demonstrates how the fTPM can become resilient to memory attacks. Sentry retrofits ARM-specific mechanisms designed for embedded systems but still present in today’s mobile devices, such as L2 cache locking or internal RAM [10]. Note that in contrast with TrustZone, Intel SGX [25] provides hardware encryption

of DRAM, which protects against memory attacks.

Side-channel attacks. Given that certain resources are shared between the secure and normal worlds, great care must be given to side-channel attacks. In contrast, a discrete TPM chip is immune to side-channel attacks that use caching, memory, or CPU because these resources are not shared with the untrusted OS.

a. Caches, memory, and CPU: The ARM TrustZone specification takes great care to reduce the likelihood of cache-based side-channel attacks for shared resources [1]. Cache-based side-channel attacks are difficult because caches are always invalidated during each transition to and from the secure world. Memory is statically partitioned between the two worlds at platform initialization time; such a static partitioning reduces the likelihood of side-channel attacks. Finally, the CPU also invalidates all its registers upon each crossing to and from the secure world.

b. Time-based attacks: The TPM 2.0 specification takes certain precautions against time-based attacks. For example, the entire cryptography subsystem of TPM 2.0 uses constant time functions – the amount of computation needed by a cryptographic function does not depend on the function’s inputs. This makes the fTPM implementation as resilient to time-based side-channel attacks as its discrete chip counterpart.

12 Discussion

Most of ARM TrustZone’s shortcomings stem from the nature of this technology: it is a standalone CPU-based security mechanism. CPU extensions alone are insufficient in many practical scenarios. As described earlier in Section 3.2, trusted systems need additional hardware support, such as support for trusted storage, secure counters, and secure peripherals.

Unfortunately, CPU designers continue to put forward CPU extensions aimed at building trusted systems that suffer from similar limitations. This section’s goal is to describe these limitations in the context of a new, up-and-coming technology called Intel Software Guard Extensions (SGX). In the absence of additional hardware support for trusted systems, our brief discussion of SGX will reveal shortcomings similar to those of TrustZone.

12.1 Intel SGX Shortcomings

Intel SGX [25] is a set of extensions to Intel processors designed to build a sandboxing mechanism for running application-level code isolated from the rest of the system. Similar to ARM TrustZone’s secure world, with Intel SGX applications can create *enclaves* protected from the OS and the rest of the platform software. All memory

allocated to an enclave is hardware encrypted (unlike the secure world in ARM). Unlike ARM TrustZone, SGX does not offer any I/O support; all interrupts are handled by the untrusted code.

SGX has numerous shortcomings for trusted systems such as the fTPM:

1. Lack of trusted storage. While code executing inside an enclave can encrypt its state, encryption cannot protect against rollback attacks. Currently, the Intel SGX specification lacks any provision to rollback protection against persisted state.

2. Lack of a secure counter. A secure counter is important when building secure systems. For example, a rollback-resilient storage system could be built using encryption and a secure counter. Unfortunately, it is difficult for a CPU to offer a secure counter without hardware assistance beyond the SGX extensions (e.g., an eMMC storage controller with an RPMB partition).

3. Lack of secure clock. SGX leaves out any specification of a secure clock. Again, it is challenging for the CPU to offer a secure clock without extra hardware.

4. Side-channel dangers. SGX enclaves only protect code running in ring 3. This means that an untrusted OS is responsible for resource management tasks, which opens up a large surface for side-channel attacks. Indeed, recent work has demonstrated a number of such attacks against Intel SGX [57].

13 Related Work

Previous efforts closest to ours are Nokia OnBoard credentials (ObC), Mobile Trusted Module (MTM), and previous software implementations of TPMs. ObC [29] is a trusted execution runtime environment leveraging Nokia’s implementation of ARM TrustZone. ObC can execute programs written in a modified variant of the LUA scripting language or written in the underlying runtime bytecode. Different scripts running in ObC are protected from each other by the underlying LUA interpreter. A more recent similar effort ported the .NET framework to TrustZone [45, 46] using techniques similar to ObC.

While the fTPM serves as the reference implementation of a firmware TPM for ARM TrustZone, ObC is a technology proprietary to Nokia. Third-parties need their code signed by Nokia to allow it to run inside TrustZone. In contrast, the fTPM offers TPM 2.0 primitives to any application. While TPM primitives are less general than a full scripting language, both researchers and industry have already used TPMs in many secure systems demonstrating its usefulness. Recognizing the TPM platform’s flexibility, ObC appears to have recently started to offer primitives more compatible with those of the TPM specification [15].

The Mobile Trusted Module (MTM) [51] is a specification similar to a TPM but aimed solely at mobile devices. Previous work investigated possible implementations of MTM for mobile devices equipped with secure hardware, such as ARM TrustZone, smartcards, and Java SecureElements [12, 13]. These related works acknowledged upfront that the limitations of ARM TrustZone for implementation MTM remain future work [12]. Unfortunately, MTMs have not gone past the specification stage in the Trusted Computing Group. As a result, we are unaware of any systems that make use of MTMs. If MTMs were to become a reality, our techniques would remain relevant in building a firmware MTM.

A more recent article presents a high-level description of the work needed to implement TPM 2.0 both in hardware and in software [34]. Like the fTPM, the article points out the need of using a replay-protected memory block partition to protect against replay attacks. However, this article appeared much later, after the fTPM was launched in mobile devices. It is unclear whether any implementation of their architecture exists.

IBM has been maintaining a software implementation of TPM 1.2 [24]. An independent effort implemented a TPM 1.2 emulator without leveraging any secure hardware [50]. This emulator was aimed at debugging scenarios and testbeds. We are unaware of efforts to integrate any of these earlier implementations into mobile devices.

Another area of related work is building virtualized TPM implementations. Virtual TPMs are needed in virtualized environments where multiple guest operating systems might want to share the physical TPM without having to trust each other. Several designs of virtual TPMs have been proposed [7, 16].

Finally, a recent survey describes additional efforts in building trusted runtime execution environments for mobile devices based on various forms of hardware, including physically uncloneable functions, smartcards, and embedded devices [4]. A recent industrial consortium called GlobalPlatform [20] has also started to put together a standard for trusted runtime execution environments on various platforms, including ARM [3].

14 Conclusions

This paper demonstrates that the limitations of CPU-based security architectures, such as ARM TrustZone, can be overcome to build software systems with security guarantees similar to those of dedicated trusted hardware. We use three different approaches to overcome these challenges: requiring additional hardware support, making design compromises without affecting security, and slightly changing command semantics.

This paper describes a software-only implementation of a TPM chip. Our software-only TPM requires no

application-level changes or changes to OS components (other than drivers). Our implementation is the reference implementation of TPM 2.0 used in millions of smartphones and tablets.

Acknowledgements We would like to thank Andrew Baumann, Weidong Cui, Roxana Geambasu, Jaeyeon Jung, and Angelos Keromytis for feedback on earlier drafts of this paper. We are also grateful to Jiajing Zhu for her help with the TPM 2.0 simulator, and numerous other collaborators who contributed to the firmware TPM effort. Finally, we would like to thank the anonymous reviewers for their feedback on the submission.

References

- [1] ARM Security Technology – Building a Secure System using TrustZone Technology. ARM Technical White Paper, 2005-2009.
- [2] Virtualization is Coming to a Platform Near You. ARM Technical White Paper, 2010-2011.
- [3] ARM. GlobalPlatform based Trusted Execution Environment and TrustZone Ready. <http://community.arm.com/servlet/JiveServlet/previewBody/8376-102-1-14233/GlobalPlatform%20based%20Trusted%20Execution%20Environment%20and%20TrustZone%20Ready%20-%20Whitepaper.pdf>.
- [4] ASOKAN, N., EKBERG, J.-E., KOSTIANEN, K., RAJAN, A., ROZAS, C., SADEGHI, A.-R., SCHULZ, S., AND WACHSMANN, C. Mobile Trusted Computing. *Proceedings of IEEE 102*, 1 (2014), 1189–1206.
- [5] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications from an Untrusted Cloud with Haven. In *Proc. of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, CO, 2014).
- [6] BECHER, M., DORNSEIF, M., AND KLEIN, C. N. FireWire - all your memory are belong to us. In *Proc. of CanSecWest Applied Security Conference* (2005).
- [7] BERGER, S., CCERES, R., GOLDMAN, K. A., PEREZ, R., SAILER, R., AND VAN DOORN, L. vtpm: Virtualizing the trusted platform module. In *Proc. of the 15th USENIX Security Symposium* (2006).
- [8] BOILEAU, A. Hit by a Bus: Physical Access Attacks with Firewire. In *Proc. of 4th Annual Ruxcon Conference* (2006).
- [9] CHEN, C., RAJ, H., SAROIU, S., AND WOLMAN, A. cTPM: A Cloud TPM for Cross-Device Trusted Applications. In *Proc. of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Seattle, WA, 2014).
- [10] COLP, P., ZHANG, J., GLEESON, J., SUNEJA, S., DE LARA, E., RAJ, H., SAROIU, S., AND WOLMAN, A. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proc. of 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Istanbul, Turkey, 2015).
- [11] DATALIGHT. What is eMMC. <http://www.datalight.com/solutions/technologies/emmc/what-is-emmc>.
- [12] DIETRICH, K., AND WINTER, J. Implementation Aspects of Mobile and Embedded Trusted Computing. *Proc. of 2nd International Conference on Trusted Computing and Trust in Information Technologies (TRUST)*, LNCS 5471 (2009), 29–44.
- [13] DIETRICH, K., AND WINTER, J. Towards Customizable, Application Specific Mobile Trusted Modules. In *Proc. of 5th ACM Workshop on Scalable Trusted Computing (STC)* (Chicago, IL, 2010).
- [14] ECRYPTFS. eCryptfs – The enterprise cryptographic filesystem for Linux. <http://ecryptfs.org/>.
- [15] EKBERG, J.-E. Mobile information security with new standards: GlobalPlatform/TCG/Nist-root-of-trust. Cyber Security and Privacy, EU Forum, 2013.
- [16] ENGLAND, P., AND LÖSER, J. Para-virtualized tpm sharing. *Proc. of 1st International Conference on Trusted Computing and Trust in Information Technologies (TRUST)*, LNCS 4968 (2008), 119–132.
- [17] EPN SOLUTIONS. Analysis tools for DDR1, DDR2, DDR3, embedded DDR and fully buffered DIMM modules. <http://www.epnsolutions.net/ddr.html>. Accessed: 2014-12-10.
- [18] FUTUREPLUS SYSTEM. DDR2 800 bus analysis probe. http://www.futureplus.com/download/datasheet/fs2334_ds.pdf, 2006.
- [19] GILBERT, P., JUNG, J., LEE, K., QIN, H., SHARKEY, D., SHETH, A., AND COX, L. P. YouProve: Authenticity and Fidelity in Mobile Sensing. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)* (Lake District, UK, 2012).
- [20] GLOBALPLATFORM. Technical Overview. <http://www.globalplatform.org/specifications.asp>.
- [21] GOGNIAT, G., WOLF, T., BURLESON, W., DIGUET, J.-P., BOSSUET, L., AND VASLIN, R. Reconfigurable hardware for high-security/high-performance embedded systems: The SAFES perspective. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 2 (2008), 144–155.
- [22] GOOGLE. The Chromium Projects. <http://www.chromium.org/developers/design-documents/tpm-usage>.
- [23] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: Cold boot attacks on encryption keys. In *Proc. of the 17th USENIX Security Symposium* (2008).
- [24] IBM. Software TPM Introduction. <http://ibmswtpm.sourceforge.net/>.
- [25] INTEL. Intel Software Guard Extensions Programming Reference. <https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf>, 2014.
- [26] JACKSON, W. Engineer shows how to crack a ‘secure’ TPM chip. <http://gcn.com/Articles/2010/02/02/Black-Hat-chip-crack-020210.aspx>, 2010.
- [27] KOCKER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. of 16th Annual International Cryptology Conference (CRYPTO)* (Santa Barbara, CA, 1996).
- [28] KOSTIAINEN, K., ASOKAN, N., AND EKBERG, J.-E. Practical Property-Based Attestation on Mobile Devices. *Proc. of 4th International Conference on Trusted Computing and Trust in Information Technologies (TRUST)*, LNCS 6470 (2011), 78–92.
- [29] KOSTIAINEN, K., EKBERG, J.-E., ASOKAN, N., AND RANTALA, A. On-board Credentials with Open Provisioning. In *Proc. of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS)* (2009).
- [30] KOTLA, R., RODEHEFFER, T., ROY, I., STUEDI, P., AND WESTER, B. Pasture: Secure Offline Data Access Using Commodity Trusted Hardware. In *Proc. of 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, 2012).
- [31] LIU, H., SAROIU, S., WOLMAN, A., AND RAJ, H. Software Abstractions for Trusted Sensors. In *Proc. of 10th International*

- Conference on Mobile Systems, Applications, and Services (MobiSys)* (Lake District, UK, 2012).
- [32] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of IEEE Symposium on Security and Privacy* (Oakland, CA, May 2010).
- [33] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)* (Glasgow, UK, 2008).
- [34] MCGILL, K. N. Trusted Mobile Devices: Requirements for a Mobile Trusted Platform Module. *Johns Hopkins Applied Physical Laboratory Technical Digest* 32, 2 (2013).
- [35] MICROSOFT. Early launch antimalware. [http://msdn.microsoft.com/en-us/library/windows/desktop/hh848061\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/hh848061(v=vs.85).aspx).
- [36] MICROSOFT. HealthAttestation CSP. <https://msdn.microsoft.com/en-us/library/dn934876%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396>.
- [37] MICROSOFT. Help protect your files with BitLocker Driver Encryption. <http://windows.microsoft.com/en-us/windows-8/using-bitlocker-drive-encryption>.
- [38] MICROSOFT. Understanding and Evaluating Virtual Smart Cards. <http://www.microsoft.com/en-us/download/details.aspx?id=29076>.
- [39] MÜLLER, T., AND SPREITZENBARTH, M. FROST - forensic recovery of scrambled telephones. In *Proc. of the International Conference on Applied Cryptography and Network Security (ACNS)* (2013).
- [40] NIST. Digital Signature Standard (DSS). <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.
- [41] PARNO, B., LORCH, J. R., DOUCEUR, J. R., MICKENS, J., AND MCCUNE, J. M. Memoir: Practical State Continuity for Protected Modules. In *Proc. of IEEE Symposium on Security and Privacy* (Oakland, CA, 2011).
- [42] PIEGDON, D. R. Hacking in physically addressable memory - a proof of concept. Presentation to the Seminar of Advanced Exploitation Techniques, 2006.
- [43] RAJ, H., ROBINSON, D., TARIQ, T., ENGLAND, P., SAROIU, S., AND WOLMAN, A. Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor. Tech. Rep. MSR-TR-2011-130, Microsoft Research, 2011.
- [44] RAJ, H., SAROIU, S., WOLMAN, A., AIGNER, R., JEREMIAH COX, A. P. E., FENNER, C., KINSHUMANN, K., LOESER, J., MATTOON, D., NYSTROM, M., ROBINSON, D., SPIGER, R., THOM, S., AND WOOTEN, D. fTPM: A Firmware-based TPM 2.0 Implementation. Tech. Rep. MSR-TR-2015-84, Microsoft, 2015.
- [45] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proc. of 12th Workshop on Mobile Computing Systems and Applications (HotMobile)* (Phoenix, AZ, 2011).
- [46] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *Proc. of 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Salt Lake City, UT, 2014).
- [47] SANTOS, N., RODRIGUES, R., GUMMADI, K. P., AND SAROIU, S. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proc. of the 21st USENIX Security Symposium* (Bellevue, WA, 2012).
- [48] SAROIU, S., AND WOLMAN, A. I Am a Sensor and I Approve This Message. In *Proc. of 11th International Workshop on Mobile Computing Systems and Applications (HotMobile)* (Annapolis, MD, 2010).
- [49] SPARKS, E., AND SMITH, S. W. TPM Reset Attack. <http://www.cs.dartmouth.edu/~pkilab/sparks/>.
- [50] STRASSER, M., AND STAMER, H. A Software-Based Trusted Platform Module Emulator. *Proc. of 1st International Conference on Trusted Computing and Trust in Information Technologies (TRUST), LNCS 4968* (2008), 33–47.
- [51] TRUSTED COMPUTING GROUP. Mobile Trusted Module Specification. http://www.trustedcomputinggroup.org/resources/mobile_phone_work_group_mobile_trusted_module_specification.
- [52] TRUSTED COMPUTING GROUP. TCPA Main Specification Version 1.1b. http://www.trustedcomputinggroup.org/files/resource_files/64795356-1D09-3519-ADAB12F595B5FCDF/TPCA_Main_TCG_Architecture_v1_1b.pdf.
- [53] TRUSTED COMPUTING GROUP. TPM 2.0 Library Specification FAQ. http://www.trustedcomputinggroup.org/resources/tpm_20_library_specification_faq.
- [54] TRUSTED COMPUTING GROUP. TPM Library Specification. http://www.trustedcomputinggroup.org/resources/tpm_library_specification.
- [55] TRUSTED COMPUTING GROUP. TPM Main Specification Level 2 Version 1.2, Revision 116. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.
- [56] WOLMAN, A., SAROIU, S., AND BAHL, V. Using Trusted Sensors to Monitor Patients' Habits. In *Proc. of 1st USENIX Workshop on Health Security and Privacy (HealthSec)* (Washington, DC, 2010).
- [57] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proc. of the 36th IEEE Symposium on Security and Privacy (Oakland)* (2015).
- [58] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proc. of Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, 2011).

Sanctum: Minimal Hardware Extensions for Strong Software Isolation

Victor Costan, Ilia Lebedev, and Srinivas Devadas
victor@costan.us, ilebedev@mit.edu, devadas@mit.edu
MIT CSAIL

Abstract

Sanctum offers the same promise as Intel’s Software Guard Extensions (SGX), namely strong provable isolation of software modules running concurrently and sharing resources, but protects against an important class of additional software attacks that infer private information from a program’s memory access patterns. Sanctum shuns unnecessary complexity, leading to a simpler security analysis. We follow a principled approach to eliminating entire attack surfaces through isolation, rather than plugging attack-specific privacy leaks. Most of Sanctum’s logic is implemented in trusted software, which does not perform cryptographic operations using keys, and is easier to analyze than SGX’s opaque microcode, which does.

Our prototype targets a Rocket RISC-V core, an open implementation that allows any researcher to reason about its security properties. Sanctum’s extensions can be adapted to other processor cores, because we do not change any major CPU building block. Instead, we add hardware at the interfaces between generic building blocks, without impacting cycle time.

Sanctum demonstrates that strong software isolation is achievable with a surprisingly small set of minimally invasive hardware changes, and a very reasonable overhead.

1 Introduction

Today’s systems rely on an operating system kernel, or a hypervisor (such as Linux or Xen, respectively) for software isolation. However **each** of the last three years (2012-2014) witnessed over 100 new security vulnerabilities in Linux [1, 11], and over 40 in Xen [2].

One may hope that formal verification methods can produce a secure kernel or hypervisor. Unfortunately, these codebases are far outside our verification capabilities: Linux and Xen have over *17 million* [6] and 150,000 [4] lines of code, respectively. In stark contrast, the seL4

formal verification effort [26] spent *20 man-years* to cover 9,000 lines of code.

Given Linux and Xen’s history of vulnerabilities and uncertain prospects for formal verification, a prudent system designer cannot include either in a TCB (trusted computing base), and must look elsewhere for a software isolation mechanism.

Fortunately, Intel’s Software Guard Extensions (SGX) [5, 36] has brought attention to the alternative of providing software isolation primitives in the CPU’s hardware. This avenue is appealing because the CPU is an unavoidable TCB component, and processor manufacturers have strong economic incentives to build correct hardware.

Unfortunately, although the SGX design includes a vast array of defenses against a variety of software and physical attacks, it fails to offer meaningful software isolation guarantees. The SGX threat model protects against all direct attacks, but excludes “side-channel attacks”, even if they can be performed via software alone.

Furthermore, our analysis [13] of SGX reveals that it is impossible for anyone but Intel to reason about SGX’s security properties, because significant implementation details are not covered by the publicly available documentation. This is a concern, as the myriad of security vulnerabilities [16, 18, 39, 50–54] in TXT [22], Intel’s previous attempt at securing remote computation, show that securing the machinery underlying Intel’s processors is incredibly challenging, even in the presence of strong economic incentives.

Our main contribution is a software isolation scheme that addresses the issues raised above: Sanctum’s isolation provably defends against known software side-channel attacks, including cache timing attacks and passive address translation attacks. Sanctum is a co-design that combines **minimal** and **minimally invasive** hardware modifications with a trusted software **security monitor** that is amenable to rigorous analysis and does not perform cryptographic operations using keys.

We achieve minimality by reusing and lightly modi-

fying existing, well-understood mechanisms. For example, our per-enclave page tables implementation uses the core’s existing page walking circuit, and requires very little extra logic. Sanctum is minimally invasive because it does not require modifying any major CPU building block. We only add hardware to the interfaces between blocks, and do not modify any block’s input or output. Our use of conventional building blocks limits the effort needed to validate a Sanctum implementation.

We demonstrate that memory access pattern attacks by malicious software can be foiled without incurring unreasonable overheads. Sanctum cores have the same clock speed as their insecure counterparts, as we do not modify the CPU core critical execution path. Using a straightforward page-coloring-based cache partitioning scheme with Sanctum adds a few percent of overhead in execution time, which is orders of magnitude lower than the overheads of the ORAM schemes [21, 43] that are usually employed to conceal memory access patterns.

All layers of Sanctum’s TCB are open-sourced at <https://github.com/pwnall/sanctum> and unencumbered by patents, trade secrets, or other similar intellectual property concerns that would disincentivize security researchers from analyzing it. Our prototype targets the Rocket Chip [29], an open-sourced implementation of the RISC-V [47, 49] instruction set architecture, which is an open standard. Sanctum’s software stack bears the MIT license.

To further encourage analysis, most of our security monitor is written in portable C++ which, once rigorously analyzed, can be used across different CPU implementations. Furthermore, even the non-portable assembly code can be reused across different implementations of the same architecture.

2 Related Work

Sanctum’s main improvement over SGX is preventing software attacks that analyze an isolated container’s memory access patterns to infer private information. We are particularly concerned with cache timing attacks [7], because they can be mounted by unprivileged software sharing a computer with the victim software.

Cache timing attacks are known to retrieve cryptographic keys used by AES [8], RSA [10], Diffie-Hellman [27], and elliptic-curve cryptography [9]. While early attacks required access to the victim’s CPU core, recent sophisticated attacks [35, 56] target the last-level cache (LLC), which is shared by all cores in a socket. Recently, [37] demonstrated a cache timing attack that uses JavaScript code in a page visited by a web browser.

Cache timing attacks observe a victim’s memory access patterns at cache line granularity. However, recent work shows that private information can be gleaned even

from the page-level memory access pattern obtained by a malicious OS that simply logs the addresses seen by its page fault handler [55].

XOM [30] introduced the idea of having sensitive code and data execute in isolated containers, and placed the OS in charge of resource allocation without trusting it. Aegis [44] relies on a trusted security kernel, handles untrusted memory, and identifies the software in a container by computing a cryptographic hash over the initial contents of the container. Aegis also computes a hash of the security kernel at boot time and uses it, together with the container’s hash, to attest a container’s identity to a third party, and to derive container keys. Unlike XOM and Aegis, Sanctum protects the memory access patterns of the software executing inside the isolation containers from software threats.

Sanctum only considers software attacks in its threat model (§ 3). Resilience against physical attacks can be added by augmenting a Sanctum processor with the countermeasures described in other secure architectures, with associated increased performance overheads. Aegis protects a container’s data when the DRAM is untrusted through memory encryption and integrity verification; these techniques were adopted and adapted by SGX. Ascend [20] and GhostRider [32] use Oblivious RAM [21] to protect a container’s memory access patterns against adversaries that can observe the addresses on the memory bus. An insight in Sanctum is that these overheads are unnecessary in a software-only threat model.

Intel’s Trusted Execution Technology (TXT) [22] is widely deployed in today’s mainstream computers, due to its approach of trying to add security to a successful CPU product. After falling victim to attacks [51, 54] where a malicious OS directed a network card to access data in the protected VM, a TXT revision introduced DRAM controller modifications that selectively block DMA transfers, which Sanctum also does.

Intel’s SGX [5, 36] adapted the ideas in Aegis and XOM to multi-core processors with a shared, coherent last-level cache. Sanctum draws heavy inspiration from SGX’s approach to memory access control, which does not modify the core’s critical execution path. We reverse-engineered and adapted SGX’s method for verifying an OS-conducted TLB shoot-down. At the same time, SGX has many security issues that are solved by Sanctum, which are stated in this paper’s introduction.

Iso-X [19] attempts to offer the SGX security guarantees, without the limitation that enclaves may only be allocated in a DRAM area that is carved off exclusively for SGX use, at boot time. Iso-X uses per-enclave page tables, like Sanctum, but its enclave page tables require a dedicated page walker. Iso-X’s hardware changes add overhead to the core’s cycle time, and do not protect against cache timing attacks.

SecureME [12] also proposes a co-design of hardware modifications and a trusted hypervisor for ensuring software isolation, but adapts the on-chip mechanisms generally used to prevent physical attacks, in order to protect applications from an untrusted OS. Just like SGX, SecureME is vulnerable to memory access pattern attacks.

The research community has brought forward various defenses against cache timing attacks. PLcache [28, 46] and the Random Fill Cache Architecture (RFill, [34]) were designed and analyzed in the context of a small region of sensitive data, and scaling them to protect a potentially large enclave without compromising performance is not straightforward. When used to isolate entire enclaves in the LLC, RFill performs at least 37%-66% worse than Sanctum.

RPcache [28, 46] trusts the OS to assign different hardware process IDs to mutually mistrusting entities, and its mechanism does not directly scale to large LLCs. The non-monopolizable cache [15] uses a well-principled partitioning scheme, but does not completely stop leakage, and relies on the OS to assign hardware process IDs. CATALYST [33] trusts the Xen hypervisor to correctly tame Intel’s Cache Allocation Technology into providing cache pinning, which can only secure software whose code and data fits into a fraction of the LLC.

Sanctum uses very simple cache partitioning [31] based on page coloring [24, 45], which has proven to have reasonable overheads. It is likely that sophisticated schemes like ZCache [40] and Vantage [41] can be combined with Sanctum’s framework to yield better performance.

3 Threat Model

Sanctum isolates the software inside an **enclave** from other software on the same computer. All outside software, including privileged system software, can only interact with an enclave via a small set of primitives provided by the security monitor. Programmers are expected to move the sensitive code in their applications into enclaves. In general, an enclave receives encrypted sensitive information from outside, decrypts the information and performs some computation on it, and then returns encrypted results to the outside world.

We assume that an attacker can compromise any operating system and hypervisor present on the computer executing the enclave, and can launch rogue enclaves. The attacker knows the target computer’s architecture and micro-architecture. The attacker can analyze passively collected data, such as page fault addresses, as well as mount active attacks, such as direct or DMA memory probing, and cache timing attacks.

Sanctum’s isolation protects the integrity and privacy of the code and data inside an enclave against any practical **software** attack that relies on observing or interacting

with the enclave software via means outside the interface provided by the security monitor. In other words, we do not protect enclaves that leak their own secrets directly (e.g., by writing to untrusted memory) or by timing their operations (e.g., by modulating their completion times). In effect, Sanctum solves the security problems that emerge from sharing a computer among mutually distrusting applications.

This distinction is particularly subtle in the context of cache timing attacks. We do not protect against attacks like [10], where the victim application leaks information via its public API, and the leak occurs even if the victim runs on a dedicated machine. We *do* protect against attacks like Flush+Reload [56], which exploit shared hardware resources to interact with the victim via methods outside its public API.

Sanctum also defeats attackers who aim to compromise an OS or hypervisor by running malicious applications and enclaves. This addresses concerns that enclaves provide new attack vectors for malware [14, 38]. We assume that the benefits of meaningful software isolation outweigh enabling a new avenue for frustrating malware detection and reverse engineering [17].

Lastly, Sanctum protects against a malicious computer owner who attempts to lie about the security monitor running on the computer. Specifically, the attacker aims to obtain an attestation stating that the computer is running an uncompromised security monitor, whereas a different monitor had been loaded in the boot process. The uncompromised security monitor must not have any known vulnerability that causes it to disclose its cryptographic keys. The attacker is assumed to know the target computer’s architecture and micro-architecture, and is allowed to run any combination of malicious security monitor, hypervisor, OS, applications and enclaves.

We do not prevent timing attacks that exploit bottlenecks in the cache coherence directory bandwidth or in the DRAM bandwidth, deferring these to future work.

Sanctum does not protect against denial-of-service (DoS) attacks by compromised system software: a malicious OS may deny service by refusing to allocate any resources to an enclave. We *do* protect against malicious enclaves attempting to DoS an uncompromised OS.

We assume correct underlying hardware, so we do not protect against software attacks that exploit hardware bugs (fault-injection attacks), such as rowhammer [25, 42].

Sanctum’s isolation mechanisms exclusively target software attacks. § 2 mentions related work that can harden a Sanctum system against some physical attacks. Furthermore, we consider software attacks that rely on sensor data to be physical attacks. For example, we do not address information leakage due to power variations, because software would require a temperature or current sensor to carry out such an attack.

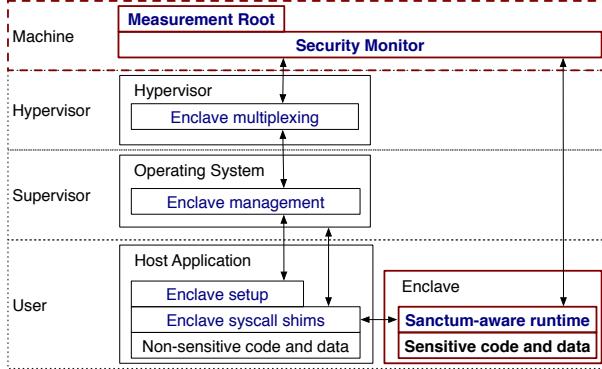


Figure 1: Software stack on a Sanctum machine; The blue text represents additions required by Sanctum. The bolded elements are in the software TCB.

4 Programming Model Overview

By design, Sanctum’s programming model deviates from SGX as little as possible, while providing stronger security guarantees. We expect that application authors will link against a Sanctum-aware runtime that abstracts away most aspects of Sanctum’s programming model. For example, C programs would use a modified implementation of the `libc` standard library. Due to space constraints, we describe the programming model assuming that the reader is familiar with SGX as described in [13].

The software stack on a Sanctum machine, shown in Figure 1, resembles the SGX stack with one notable exception: SGX’s microcode is replaced by a trusted software component, the **security monitor**, which is protected from compromised system software, as it runs at the highest privilege level (machine level in RISC-V).

We relegate the management of computation resources, such as DRAM and execution cores, to untrusted system software (as does SGX). In Sanctum, the security monitor checks the system software’s allocation decisions for correctness and commits them into the hardware’s configuration registers. For simplicity, we refer to the software that manages resources as an *OS (operating system)*, even though it may be a combination of a hypervisor and a guest OS kernel.

An enclave stores its code and private data in parts of DRAM that have been allocated by the OS exclusively for the enclave’s use (as does SGX), which are collectively called **the enclave’s memory**. Consequently, we refer to the regions of DRAM that are not allocated to any enclave as **OS memory**. The security monitor tracks DRAM ownership, and ensures that no piece of DRAM is assigned to more than one enclave.

Each Sanctum enclave uses a range of virtual memory addresses (EVRANGE) to access its memory. The enclave’s memory is mapped by the enclave’s own page ta-

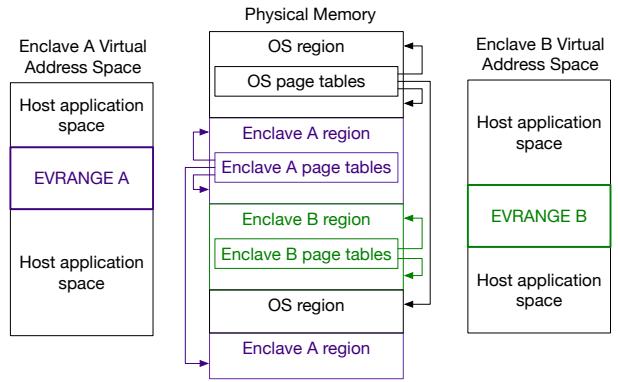


Figure 2: Per-enclave page tables

bles, which are stored in the enclave’s memory (Figure 2). This makes private the page table dirty and accessed bits, which can reveal memory access patterns at page granularity. Exposing an enclave’s page tables to the untrusted OS leaves the enclave vulnerable to attacks such as [55].

The enclave’s virtual address space outside EVRANGE is used to access its host application’s memory, via the page tables set up by the OS. Sanctum’s hardware extensions implement dual page table lookup (§ 5.2), and make sure that an enclave’s page tables can only point into the enclave’s memory, while OS page tables can only point into OS memory (§ 5.3).

Sanctum supports multi-threaded enclaves, and enclaves must appropriately provision for thread state data structures. Enclave threads, like their SGX cousins, run at the lowest privilege level (user level in RISC-V), meaning a malicious enclave cannot compromise the OS. Specifically, enclaves may not execute privileged instructions; address translations that use OS page tables generate page faults when accessing supervisor pages.

The per-enclave metadata used by the security monitor is stored in dedicated DRAM regions (**metadata regions**), each managed at the page level by the OS, and each includes a page map that is used by the security monitor to verify the OS’ decisions (much like the EPC and EPCM in SGX, respectively). Unlike SGX’s EPC, the metadata region pages only store enclave and thread metadata. Figure 3 shows how these structures are weaved together.

Sanctum considers system software to be untrusted, and governs transitions into and out of enclave code. An enclave’s host application **enters an enclave** via a security monitor call that locks a thread state area, and transfers control to its entry point. After completing its intended task, the enclave code **exits** by asking the monitor to unlock the thread’s state area, and transfer control back to the host application.

Enclaves cannot make system calls directly: we cannot trust the OS to restore an enclave’s execution state, so the

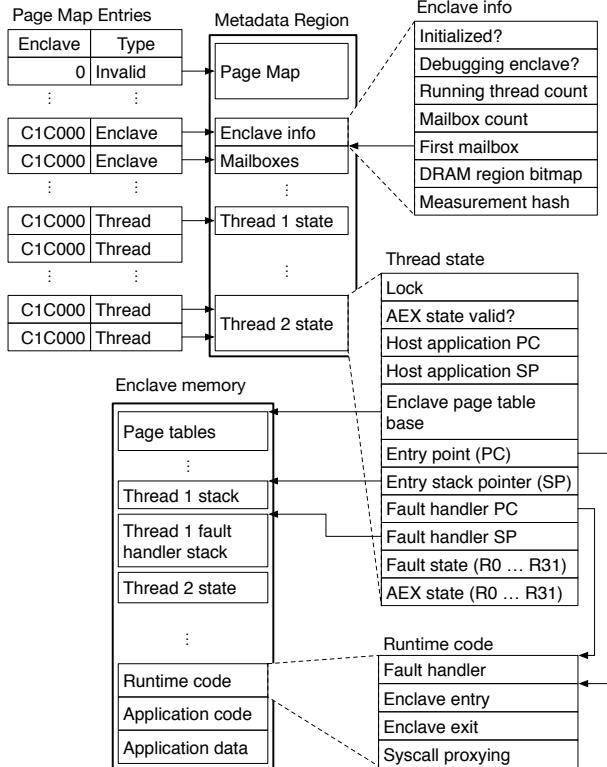


Figure 3: Enclave layout and data structures

enclave’s runtime must ask the host application to proxy syscalls such as file system and network I/O requests.

Sanctum’s security monitor is the first responder for interrupts: an interrupt received during enclave execution causes an *asynchronous enclave exit* (AEX), whereby the monitor saves the core’s registers in the current thread’s AEX state area, zeroes the registers, exits the enclave, and dispatches the interrupt as if it was received by the code entering the enclave.

Unlike SGX, resuming enclave execution after an AEX means re-entering the enclave using its normal entry point, and having the enclave’s code ask the security monitor to restore the pre-AEX execution state. Sanctum enclaves are aware of asynchronous exits so they can implement security policies. For example, an enclave thread that performs time-sensitive work, such as periodic I/O, may terminate itself if it ever gets preempted by an AEX.

The security monitor configures the CPU to dispatch all faults occurring within an enclave directly to the enclave’s designated fault handler, which is expected to be implemented by the enclave’s runtime (SGX sanitizes and dispatches faults to the OS). For example, a libc runtime would translate faults into UNIX signals which, by default, would exit the enclave. It is possible, though not advisable for performance reasons (§ 6.3), for a runtime to handle page faults and implement demand paging

securely, and robust against the attacks described in [55].

Unlike SGX, we isolate each enclave’s data throughout the system’s cache hierarchy. The security monitor flushes per-core caches, such as the L1 cache and the TLB, whenever a core jumps between enclave and non-enclave code. *Last-level cache* (LLC) isolation is achieved by a simple partitioning scheme supported by Sanctum’s hardware extensions (§ 5.1).

Sanctum’s strong isolation yields a simple security model for application developers: *all computation that executes inside an enclave, and only accesses data inside the enclave, is protected from any attack mounted by software outside the enclave*. All communication with the outside world, including accesses to non-enclave memory, is subject to attacks.

We assume that the enclave runtime implements the security measures needed to protect the enclave’s communication with other software modules. For example, any algorithm’s memory access patterns can be protected by ensuring that the algorithm only operates on enclave data. The runtime can implement this protection simply by copying any input buffer from non-enclave memory into the enclave before computing on it.

The enclave runtime can use Native Client’s approach [57] to ensure that the rest of the enclave software only interacts with the host application via the runtime to mitigate potential security vulnerabilities in enclave software.

The lifecycle of a Sanctum enclave closely resembles the lifecycle of its SGX equivalent. An enclave is created when its host application performs a system call asking the OS to create an enclave from a dynamically loadable module (.so or .d11 file). The OS invokes the security monitor to assign DRAM resources to the enclave, and to load the initial code and data pages into the enclave. Once all the pages are loaded, the enclave is marked as initialized via another security monitor call.

Our software attestation scheme is a simplified version of SGX’s scheme, and reuses a subset of its concepts. The data used to initialize an enclave is cryptographically hashed, yielding the enclave’s *measurement*. An enclave can invoke a secure inter-enclave messaging service to send a message to a privileged *attestation enclave* that can access the security monitor’s attestation key, and produces the attestation signature.

5 Hardware Modifications

5.1 LLC Address Input Transformation

Figure 4 depicts a physical address in a toy computer with 32-bit virtual addresses and 21-bit physical addresses, 4,096-byte pages, a set-associative LLC with 512 sets and 64-byte lines, and 256 KB of DRAM.

The location where a byte of data is cached in the

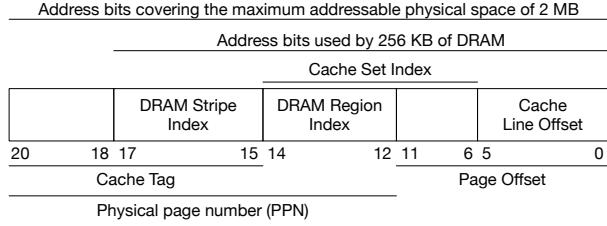


Figure 4: Interesting bit fields in a physical address

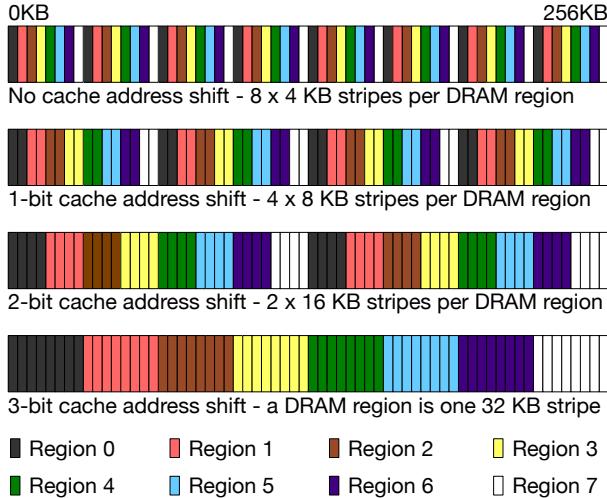


Figure 5: Address shift for contiguous DRAM regions

LLC depends on the low-order bits in the byte's physical address. The *set index* determines which of the LLC lines can cache the line containing the byte, and the *line offset* locates the byte in its cache line. A virtual address's low-order bits make up its *page offset*, while the other bits are its *virtual page number* (VPN). Address translation leaves the page offset unchanged, and translates the VPN into a *physical page number* (PPN), based on the mapping specified by the page tables.

We define the **DRAM region index** in a physical address as the intersection between the PPN bits and the cache index bits. This is the maximal set of bits that impact cache placement *and* are determined by privileged software via page tables. We define a **DRAM region** to be the subset of DRAM with addresses having the same DRAM region index. In Figure 4, for example, address bits [14...12] are the DRAM region index, dividing the physical address space into 8 DRAM regions.

In a typical system without Sanctum's hardware extensions, DRAM regions are made up of multiple continuous **DRAM stripes**, where each stripe is exactly one page long. The top of Figure 5 drives this point home, by showing the partitioning of our toy computer's 256 KB of DRAM into DRAM regions. The fragmentation of

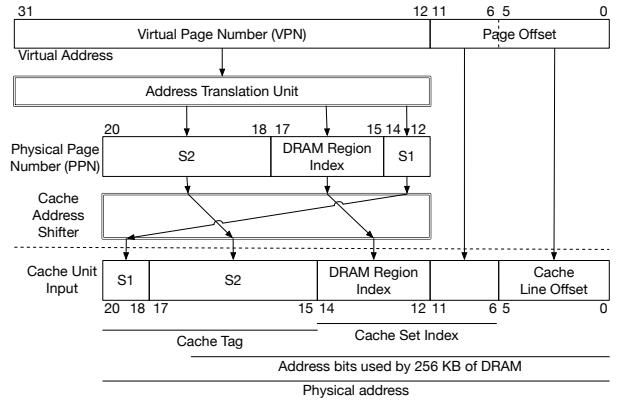


Figure 6: Cache address shifter, 3 bit PPN rotation

DRAM regions makes it difficult for the OS to allocate contiguous DRAM buffers, which are essential to the efficient DMA transfers used by high performance devices. In our example, if the OS only owns 4 DRAM regions, the largest contiguous DRAM buffer it can allocate is 16 KB.

We observed that, up to a certain point, circularly shifting (rotating) the PPN of a physical address to the right by one bit, before it enters the LLC, doubles the size of each DRAM stripe and halves the number of stripes in a DRAM region, as illustrated in Figure 5.

Sanctum takes advantage of this effect by adding a **cache address shifter** that circularly shifts the PPN to the right by a certain amount of bits, as shown in Figures 6 and 7. In our example, configuring the cache address shifter to rotate the PPN by 3 yields contiguous DRAM regions, so an OS that owns 4 DRAM regions could hypothetically allocate a contiguous DRAM buffer covering half of the machine's DRAM.

The cache address shifter's configuration depends on the amount of DRAM present in the system. If our example computer could have 128 KB - 1 MB of DRAM, its cache address shifter must support shift amounts from 2 to 5. Such a shifter can be implemented via a 3-position variable shifter circuit (series of 8-input MUXes), and a fixed shift by 2 (no logic). Alternatively, in systems with known DRAM configuration (embedded, SoC, etc.), the shift amount can be fixed, and implemented with no logic.

5.2 Page Walker Input

Sanctum's per-enclave page tables require an enclave page table base register `eptbr` that stores the physical address of the currently running enclave's page tables, and has similar semantics to the page table base register `ptbr` pointing to the operating system-managed page tables. These registers may only be accessed by the Sanctum security monitor, which provides an API call for the OS

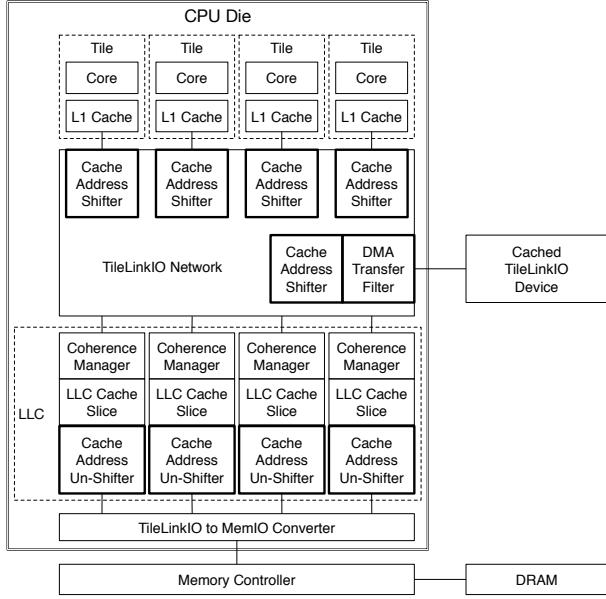


Figure 7: Sanctum’s cache address shifter and DMA transfer filter logic in the context of a Rocket uncore

to modify `ptbr`, and ensures that `eptbr` always points to the current enclave’s page tables.

The circuitry handling TLB misses switches between `ptbr` and `eptbr` based on two registers that indicate the current enclave’s EVRANGE, namely `evbase` (enclave virtual address space base) and `evmask` (enclave virtual address space mask). When a TLB miss occurs, the circuit in Figure 8 selects the appropriate page table base by ANDing the faulting virtual address with the mask register and comparing the output against the base register. Depending on the comparison result, either `eptbr` or `ptbr` is forwarded to the page walker as the page table base address.

5.3 Page Walker Memory Accesses

In modern high-speed CPUs, address translation is performed by a hardware **page walker** that traverses the page tables when a TLB miss occurs. The page walker’s latency greatly impacts the CPU’s performance, so it is implemented as a finite-state machine (FSM) that reads page table entries by issuing DRAM read requests using physical addresses, over a dedicated bus to the L1 cache.

Unsurprisingly, page walker modifications require a lot of engineering effort. At the same time, Sanctum’s security model demands that the page walker only references enclave memory when traversing the enclave page tables, and only references OS memory when translating the OS page tables. Fortunately, we can satisfy these requirements without modifying the FSM. Instead, the security monitor configures the circuit in Figure 9 to ensure that

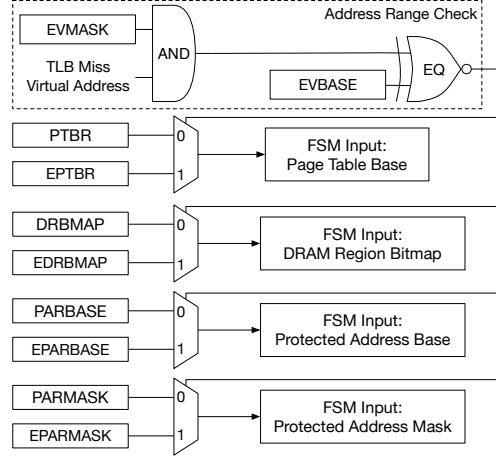


Figure 8: Page walker input for per-enclave page tables

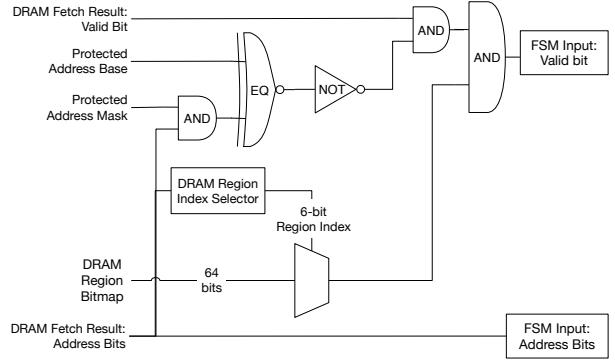


Figure 9: Hardware support for per-enclave page tables: check page table entries fetched by the page walker.

the page tables only point into allowable memory.

Sanctum’s security monitor must guarantee that `ptbr` points into an OS DRAM region, and `eptbr` points into a DRAM region owned by the enclave. This secures the page walker’s initial DRAM read. The circuit in Figure 9 receives each page table entry fetched by the FSM, and sanitizes it before it reaches the page walker FSM.

The security monitor configures the set of DRAM regions that page tables may reference by writing to a DRAM region bitmap (`drbmap`) register. The sanitization circuitry extracts the DRAM region index from the address in the page table entry, and looks it up in the DRAM region bitmap. If the address does not belong to an allowable DRAM region, the sanitization logic forces the page table entry’s valid bit to zero, which will cause the page walker FSM to abort the address translation and signal a page fault.

Sanctum’s security monitor and its attestation key are stored in DRAM regions allocated to the OS. For security reasons, the OS must not be able to modify the monitor’s

code, or to read the attestation key. Sanctum extends the page table entry transformation described above to implement a Protected Address Range (PAR) for each set of page tables.

Each PAR is specified using a base register (`parbase`) register and a mask register (`parmask`) with the same semantics as the variable Memory Type Range registers (MTRRs) in the x86 architecture. The page table entry sanitization logic in Sanctum's hardware extensions checks if each page table entry points into the PAR by ANDing the entry's address with the PAR mask and comparing the result with the PAR base. If a page table entry is seen with a protected address, its valid bit is cleared, forcing a page fault.

The above transformation allows the security monitor to set up a memory range that cannot be accessed by other software, and which can be used to securely store the monitor's code and data. Entry invalidation ensures no page table entries are fetched from the protected range, which prevents the page walker FSM from modifying the protected region by setting accessed and dirty bits.

All registers above are replicated, as Sanctum maintains separate OS and enclave page tables. The security monitor sets up a protected range in the OS page tables to isolate its own code and data structures (most importantly its private attestation key) from a malicious OS.

Figure 10 shows Sanctum’s logic inserted between the page walker and the cache unit that fetches page table entries.

5.4 DMA Transfer Filtering

We whitelist a DMA-safe DRAM region instead of following SGX’s blacklist approach. Specifically, Sanctum adds two registers (a base, dmarbase and an AND mask, dmarmask) to the DMA arbiter (memory controller). The range check circuit shown in Figure 8 compares each DMA transfer’s start and end addresses against the allowed DRAM range, and the DMA arbiter drops transfers that fail the check.

6 Software Design

Sanctum's chain of trust, discussed in § 6.1, diverges significantly from SGX. We replace SGX's microcode with a software security monitor that runs at a higher privilege level than the hypervisor and the OS. On RISC-V, the security monitor runs at machine level. Our design only uses one privileged enclave, the signing enclave, which behaves similarly to SGX's Quoting Enclave.

6.1 Attestation Chain of Trust

Sanctum has three pieces of trusted software: the measurement root, which is burned in on-chip ROM, the security monitor (§ 6.2), which must be stored alongside

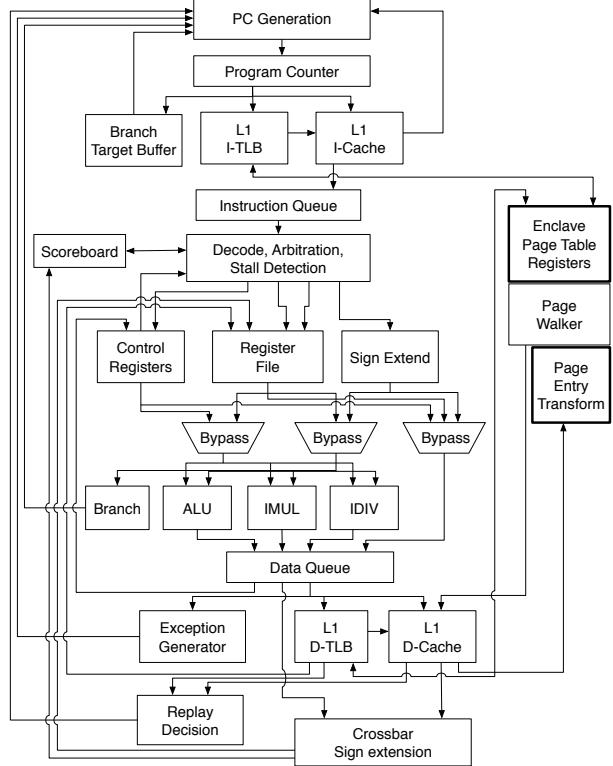


Figure 10: Sanctum’s page entry transformation logic in the context of a Rocket core

the computer's firmware (usually in flash memory), and the signing enclave, which can be stored in any untrusted storage that the OS can access.

We expect the trusted software to be amenable to rigorous analysis: our implementation of a security monitor for Sanctum is written with verification in mind, and has fewer than 5 kloc of C++, including a subset of the standard library and the cryptography for enclave attestation.

6.1.1 The Measurement Root

The measurement root (`mroot`) is stored in a ROM at the top of the physical address space, and covers the reset vector. Its main responsibility is to compute a cryptographic hash of the security monitor and generate a monitor attestation key pair and certificate based on the monitor's hash, as shown in Figure 11.

The security monitor is expected to be stored in non-volatile memory (such as an SPI flash chip) that can respond to memory I/O requests from the CPU, perhaps via a special mapping in the computer's chipset. When `mroot` starts executing, it computes a cryptographic hash over the security monitor. `mroot` then reads the processor's key derivation secret, and derives a symmetric key based on the monitor's hash. `mroot` will eventually hand

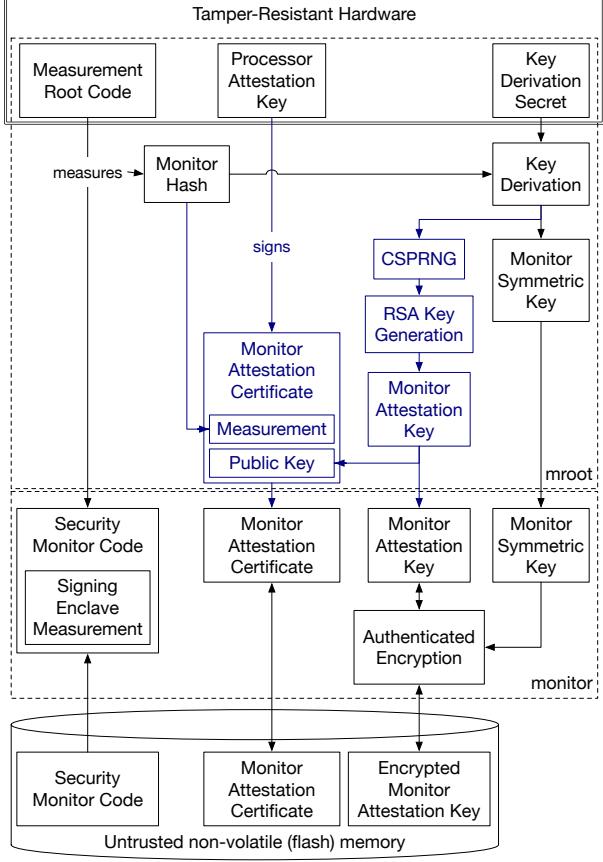


Figure 11: Sanctum’s root of trust is a measurement root routine burned into the CPU’s ROM. This code reads the security monitor from flash memory and generates an attestation key and certificate based on the monitor’s hash. Asymmetric key operations, colored in blue, are only performed the first time a monitor is used on a computer.

down the key to the monitor.

The security monitor contains a header that includes the location of an attestation key existence flag. If the flag is not set, the measurement root generates a monitor attestation key pair, and produces a monitor attestation certificate by signing the monitor’s public attestation key with the processor’s private attestation key. The monitor attestation certificate includes the monitor’s hash.

`mroot` generates a symmetric key for the security monitor so it may encrypt its private attestation key and store it in the computer’s SPI flash memory chip. When writing the key, the monitor also sets the monitor attestation key existence flag, instructing future boot sequences not to regenerate a key. The public attestation key and certificate can be stored unencrypted in any untrusted memory.

Before handing control to the monitor, `mroot` sets a lock that blocks any software from reading the processor’s symmetric key derivation seed and private key until a reset

occurs. This prevents a malicious security monitor from deriving a different monitor’s symmetric key, or from generating a monitor attestation certificate that includes a different monitor’s measurement hash.

The symmetric key generated for the monitor is similar in concept to the Seal Keys produced by SGX’s key derivation process, as it is used to securely store a secret (the monitor’s attestation key) in untrusted memory, in order to avoid an expensive process (asymmetric key attestation and signing). Sanctum’s key derivation process is based on the monitor’s measurement, so a given monitor is guaranteed to get the same key across power cycles. The cryptographic properties of the key derivation process guarantee that a malicious monitor cannot derive the symmetric key given to another monitor.

6.1.2 The Signing Enclave

In order to avoid timing attacks, the security monitor does not compute attestation signatures directly. Instead, the signing algorithm is executed inside a signing enclave, which is a security monitor module that executes in an enclave environment, so it is protected by the same isolation guarantees that any other Sanctum enclave enjoys.

The signing enclave receives the monitor’s private attestation key via an API call. When the security monitor receives the call, it compares the calling enclave’s measurement with the known measurement of the signing enclave. Upon a successful match, the monitor copies its attestation key into enclave memory using a data-independent sequence of memory accesses, such as `memcpy`. This way, the monitor’s memory access pattern does not leak the private attestation key.

Sanctum’s signing enclave authenticates another enclave on the computer and securely receives its attestation data using mailboxes (§ 6.2.5), a simplified version of SGX’s local attestation (reporting) mechanism. The enclave’s measurement and attestation data are wrapped into a software attestation signature that can be examined by a remote verifier. Figure 12 shows the chain of certificates and signatures in an instance of software attestation.

6.2 Security Monitor

The security monitor receives control after `mroot` finishes setting up the attestation measurement chain.

The monitor provides API calls to the OS and enclaves for **DRAM region allocation** and **enclave management**. The monitor guards sensitive registers, such as the page table base register (`ptbr`) and the allowed DMA range (`dmabase` and `dmarmask`). The OS can set these registers via monitor calls that ensure the register values are consistent with the current DRAM region allocation.

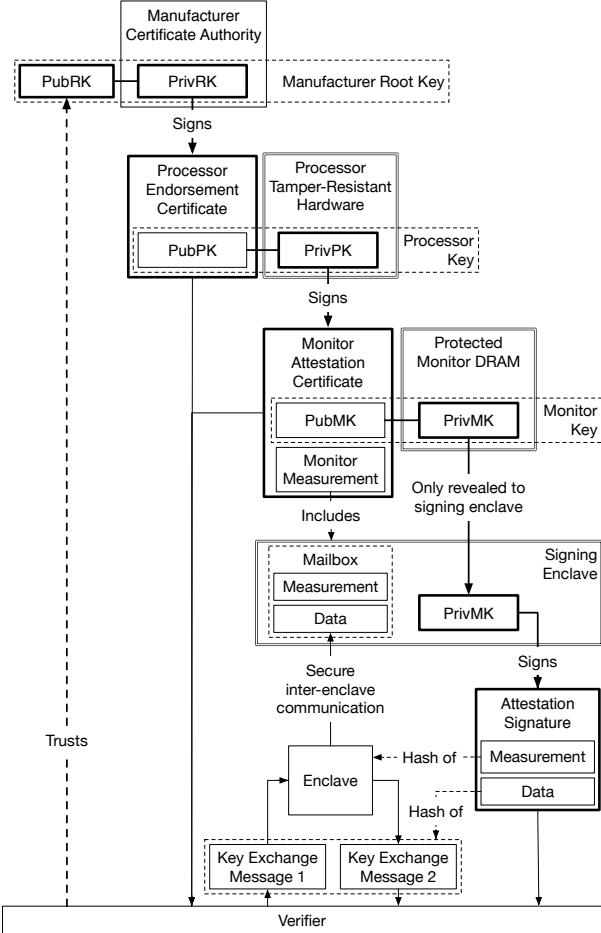


Figure 12: The certificate chain behind Sanctum’s software attestation signatures

6.2.1 DRAM Regions

Figure 13 shows the DRAM region allocation state transition diagram. After the system boots up, all DRAM regions are allocated to the OS, which can free up DRAM regions so it can re-assign them to enclaves or to itself. A DRAM region can only become free after it is blocked by its owner, which can be the OS or an enclave. While a DRAM region is blocked, any address translations mapping to it cause page faults, so no new TLB entries will be created for that region. Before the OS frees the blocked region, it must flush all the cores’ TLBs, to remove any stale entries for the region.

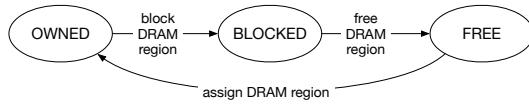


Figure 13: DRAM region allocation states and API calls

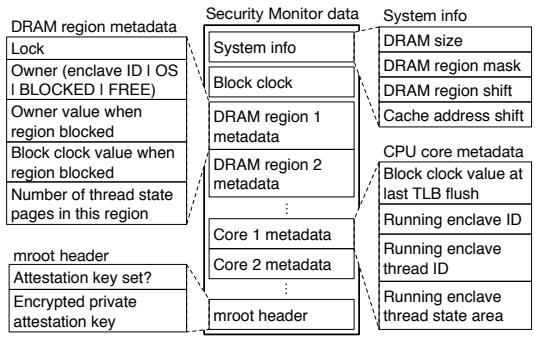


Figure 14: Security monitor data structures

The monitor ensures that the OS performs TLB shootdowns, using a global *block clock*. When a region is blocked, the block clock is incremented, and the current block clock value is stored in the metadata associated with the DRAM region (shown in Figure 3). When a core’s TLB is flushed, that core’s flush time is set to the current block clock value. When the OS asks the monitor to free a blocked DRAM region, the monitor verifies that no core’s flush time is lower than the block clock value stored in the region’s metadata. As an optimization, freeing a region owned by an enclave only requires TLB flushes on the cores running that enclave’s threads. No other core can have TLB entries for the enclave’s memory.

The region blocking mechanism guarantees that when a DRAM region is assigned to an enclave or the OS, no stale TLB mappings associated with the DRAM region exist. The monitor uses the MMU extensions described in § 5.2 and § 5.3 to ensure that once a DRAM region is assigned, no software other than the region’s owner may create TLB entries pointing inside the DRAM region. Together, these mechanisms guarantee that the DRAM regions allocated to an enclave cannot be accessed by the operating system or by another enclave.

6.2.2 Metadata Regions

Since the security monitor sits between the OS and enclave, and its APIs can be invoked by both sides, it is an easy target for timing attacks. We prevent these attacks with a straightforward policy that states the security monitor is never allowed to access enclave data, and is not allowed to make memory accesses that depend on the attestation key material. The rest of the data handled by the monitor is derived from the OS’ actions, so it is already known to the OS.

A rather obvious consequence of the policy above is that after the security monitor boots the OS, it cannot perform any cryptographic operations that use keys. For example, the security monitor cannot compute an attestation signature directly, and defers that operation to a sign-

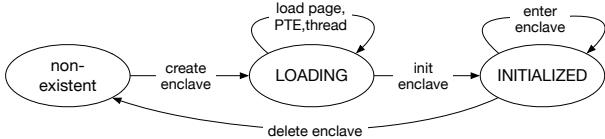


Figure 15: Enclave states and enclave management API calls

ing enclave (§ 6.1.2). While it is possible to implement some cryptographic primitives without performing data-dependent accesses, the security and correctness proofs behind these implementations are non-trivial. For this reason, Sanctum avoids depending on any such implementation.

A more subtle aspect of the access policy outlined above is that the metadata structures that the security monitor uses to operate enclaves cannot be stored in DRAM regions owned by enclaves, because that would give the OS an indirect method of accessing the LLC sets that map to enclave’s DRAM regions, which could facilitate a cache timing attack.

For this reason, the security monitor requires the OS to set aside at least one DRAM region for enclave metadata before it can create enclaves. The OS has the ability to free up the metadata DRAM region, and regain the LLC sets associated with it, if it predicts that the computer’s workload will not involve enclaves.

Each DRAM region that holds enclave metadata is managed independently from the other regions, at page granularity. The first few pages of each region contain a page map that tracks the enclave that tracks the usage of each metadata page, specifically the enclave that it is assigned to, and the data structure that it holds.

Each metadata region is like an EPC region in SGX, with the exception that our metadata regions only hold special pages, like Sanctum’s equivalent of SGX’s Secure Enclave Control Structure (SECS) and the Thread Control Structure (TCS). These structures will be described in the following sections.

The data structures used to store Sanctum’s metadata can span multiple pages. When the OS allocates such a structure in a metadata region, it must point the monitor to a sequence of free pages that belong to the same DRAM region. All the pages needed to represent the structure are allocated and released in one API call.

6.2.3 Enclave Lifecycle

The lifecycle of a Sanctum enclave is very similar to that of its SGX counterparts, as shown in Figure 15.

The OS creates an enclave by issuing a *create enclave* call that creates the enclave metadata structure, which is Sanctum’s equivalent of the SECS. The enclave metadata

structure contains an array of mailboxes whose size is established at enclave creation time, so the number of pages required by the structure varies from enclave to enclave. § 6.2.5 describes the contents and use of mailboxes.

The *create enclave* API call initializes the enclave metadata fields shown in Figure 3, and places the enclave in the LOADING state. While the enclave is in this state, the OS sets up the enclave’s initial state via monitor calls that assign DRAM regions to the enclave, create hardware threads and page table entries, and copy code and data into the enclave. The OS then issues a monitor call to transition the enclave to the INITIALIZED state, which finalizes its measurement hash. The application hosting the enclave is now free to run enclave threads.

Sanctum stores a measurement hash for each enclave in its metadata area, and updates the measurement to account for every operation performed on an enclave in the LOADING state. The policy described in § 6.2.2 does not apply to the secure hash operations used to update enclave’s measurement, because all the data used to compute the hash is already known to the OS.

Enclave metadata is stored in a metadata region (§ 6.2.2), so it can only be accessed by the security monitor. Therefore, the metadata area can safely store public information with integrity requirements, such as the enclave’s measurement hash.

While an OS loads an enclave, it is free to map the enclave’s pages, but the monitor maintains its page tables ensuring all entries point to non-overlapping pages in DRAM owned by the enclave. Once an enclave is initialized, it can inspect its own page tables and abort if the OS created undesirable mappings. Simple enclaves do not require specific mappings. Complex enclaves are expected to communicate their desired mappings to the OS via out-of-band metadata not covered by this work.

Our monitor ensures that page tables do not overlap by storing the last mapped page’s physical address in the enclave’s metadata. To simplify the monitor, a new mapping is allowed if its physical address is greater than that of the last, constraining the OS to map an enclave’s DRAM pages in monotonically increasing order.

6.2.4 Enclave Code Execution

Sanctum closely follows the threading model of SGX enclaves. Each CPU core that executes enclave code uses a thread metadata structure, which is our equivalent of SGX’s TCS combined with SGX’s State Save Area (SSA). Thread metadata structures are stored in a DRAM region dedicated to enclave metadata in order to prevent a malicious OS from mounting timing attacks against an enclave by causing AEXes on its threads. Figure 16 shows the lifecycle of a thread metadata structure.

The OS turns a sequence of free pages in a metadata

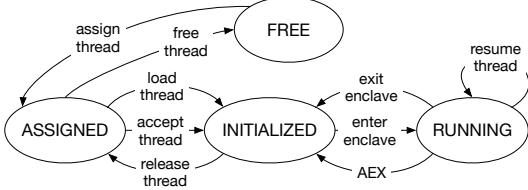


Figure 16: Enclave thread metadata structure states and thread-related API calls

region into an uninitialized thread structure via an *allocate thread* monitor call. During enclave loading, the OS uses a *load thread* monitor call to initialize the thread structure with data that contributes to the enclave’s measurement. After an enclave is initialized, it can use an *accept thread* monitor call to initialize its thread structure.

The application hosting an enclave starts executing enclave code by issuing an *enclave enter* API call, which must specify an initialized thread structure. The monitor honors this call by configuring Sanctum’s hardware extensions to allow access to the enclave’s memory, and then by loading the program counter and stack pointer registers from the thread’s metadata structure. The enclave’s code can return control to the hosting application voluntarily, by issuing an *enclave exit* API call, which restores the application’s PC and SP from the thread state area and sets the API call’s return value to `ok`.

When performing an AEX, the security monitor atomically tests-and-sets the *AEX state valid* flag in the current thread’s metadata. If the flag is clear, the monitor stores the core’s execution state in the thread state’s AEX area. Otherwise, the enclave thread was resuming from an AEX, so the monitor does not change the AEX area. When the host application re-enters the enclave, it will resume from the previous AEX. This reasoning avoids the complexity of SGX’s state stack.

If an interrupt occurs while the enclave code is executing, the security monitor’s exception handler performs an AEX, which sets the API call’s return value to `async_exit`, and invokes the standard interrupt handling code. After the OS handles the interrupt, the enclave’s host application resumes execution, and re-executes the *enter enclave* API call. The enclave’s thread initialization code examines the saved thread state, and seeing that the thread has undergone an AEX, issues a *resume thread* API call. The security monitor restores the enclave’s registers from the thread state area, and clears the AEX flag.

6.2.5 Mailboxes

Sanctum’s software attestation process relies on *mailboxes*, which are a simplified version of SGX’s local attestation mechanism. We could not follow SGX’s approach

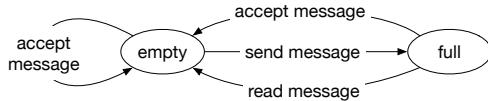


Figure 17: Mailbox states and security monitor API calls related to inter-enclave communication

because it relies on key derivation and MAC algorithms, and our timing attack avoidance policy (§ 6.2.2) states that the security monitor is not allowed to perform cryptographic operations that use keys.

Each enclave’s metadata area contains an array of mailboxes, whose size is specified at enclave creation time, and covered by the enclave’s measurement. Each mailbox goes through the lifecycle shown in Figure 17.

An enclave that wishes to receive a message in a mailbox, such as the signing enclave, declares its intent by performing an *accept message* monitor call. The API call is used to specify the mailbox that will receive the message, and the identity of the enclave that is expected to send the message.

The sending enclave, which is usually the enclave wishing to be authenticated, performs a *send message* call that specifies the identity of the receiving enclave, and a mailbox within that enclave. The monitor only delivers messages to mailboxes that expect them. At enclave initialization, the expected sender for all mailboxes is an invalid value (all zeros), so the enclave will not receive messages until it calls *accept message*.

When the receiving enclave is notified via an out-of-band mechanism that it has received a message, it issues a *read message* call to the monitor, which moves the message from the mailbox into the enclave’s memory. If the API call succeeds, the receiving enclave is assured that the message was sent by the enclave whose identity was specified in the *accept message* call.

Enclave mailboxes are stored in metadata regions (§ 6.2.2), which cannot be accessed by any software other than the security monitor. This guarantees the privacy, integrity, and freshness of the messages sent via the mailbox system.

Our mailbox design has the downside that both the sending and receiving enclave need to be alive in DRAM in order to communicate. By comparison, SGX’s local attestation can be done asynchronously. In return, mailboxes do not require any cryptographic operations, and have a much simpler correctness argument.

6.2.6 Multi-Core Concurrency

The security monitor is highly concurrent, with fine-grained locks. API calls targeting two different enclaves may be executed in parallel on different cores. Each

DRAM region has a lock guarding that region’s metadata. An enclave is guarded by the lock of the DRAM region holding its metadata. Each thread metadata structure also has a lock guarding it, which is acquired when the structure is accessed, but also while the metadata structure is used by a core running enclave code. Thus, the *enter enclave* call acquires a slot lock, which is released by an *enclave exit* call or by an AEX.

We avoid deadlocks by using a form of optimistic locking. Each monitor call attempts to acquire all the locks it needs via atomic test-and-set operations, and errors with a `concurrent_call` code if any lock is unavailable.

6.3 Enclave Eviction

General-purpose software can be enclaved without source code changes, provided that it is linked against a runtime (e.g., *libc*) modified to work with Sanctum. Any such runtime would be included in the TCB.

The Sanctum design allows the operating system to over-commit physical memory allocated to enclaves, by collaborating with an enclave to page some of its DRAM regions to disk. Sanctum does not give the OS visibility into enclave memory accesses, in order to prevent private information leaks, so the OS must decide the enclave whose DRAM regions will be evicted based on other activity, such as network I/O, or based on a business policy, such as Amazon EC2’s spot instances.

Once a victim enclave has been decided, the OS asks the enclave to block a DRAM region (cf. Figure 13), giving the enclave an opportunity to rearrange data in its RAM regions. DRAM region management can be transparent to the programmer if handled by the enclave’s runtime. The presented design requires each enclave to always occupy at least one DRAM region, which contains enclave data structures and the memory management code described above. Evicting all of a live enclave’s memory requires an entirely different scheme that is deferred to future work.

The security monitor does not allow the OS to forcibly reclaim a single DRAM region from an enclave, as doing so would leak memory access patterns. Instead, the OS can delete an enclave, after stopping its threads, and reclaim all its DRAM regions. Thus, a small or short-running enclave may well refuse DRAM region management requests from the OS, and expect the OS to delete and restart it under memory pressure.

To avoid wasted work, large long-running enclaves may elect to use demand paging to overcommit their DRAM, albeit with the understanding that demand paging leaks page-level access patterns to the OS. Securing this mechanism requires the enclave to obfuscate its page faults via periodic I/O using oblivious RAM techniques, as in the Ascend processor [20], applied at page rather than cache line granularity, and with integrity verification.

This carries a high overhead: even with a small chance of paging, an enclave must generate periodic page faults, and access a large set of pages at each period. Using an analytic model, we estimate the overhead to be upwards of 12ms per page per period for a high-end 10K RPM drive, and 27ms for a value hard drive. Given the number of pages accessed every period grows with an enclave’s data size, the costs are easily prohibitive. While SSDs may alleviate some of this prohibitive overhead, and will be investigated in future work, currently Sanctum focuses on securing enclaves without demand paging.

Enclaves that perform other data-dependent communication, such as targeted I/O into a large database file, must also use the periodic oblivious I/O to obfuscate their access patterns from the operating system. These techniques are independent of application business logic, and can be provided by libraries such as database access drivers.

7 Security Argument

Our security argument rests on two pillars: the enclave isolation enforced by the security monitor, and the guarantees behind the software attestation signature. This section outlines correctness arguments for each of these pillars.

Sanctum’s isolation primitives protect enclaves from outside software that attempts to observe or interact with the enclave software via means outside the interface provided by the security monitor. We prevent direct attacks by ensuring that the memory owned by an enclave can only be accessed by that enclave’s software. More subtle attacks are foiled by also isolating the structures used to access the enclave’s memory, such as the enclave’s page tables and the caches that hold enclave data.

7.1 Protection Against Direct Attacks

The correctness proof for Sanctum’s DRAM isolation can be divided into two sub-proofs that cover the hardware and software sides of the system. First, we need to prove that the page walker modifications described in § 5.2 and § 5.3 behave according to their descriptions. Thanks to the small sizes of the circuits involved, this sub-proof can be accomplished by simulating the circuits for all logically distinct input cases. Second, we must prove that the security monitor configures Sanctum’s extended page walker registers in a way that prevents direct attacks on enclaves. This part of the proof is significantly more complex, but it follows the same outline as the proof for SGX’s memory access protection presented in [13].

The proof revolves around a main invariant stating that all TLB entries in every core are consistent with the programming model described in § 4. The invariant breaks down into three cases that match [13], after substituting DRAM regions for pages.

7.2 Protection Against Subtle Attacks

Sanctum also protects enclaves from software attacks that attempt to exploit side channels to obtain information indirectly. We focus on proving that Sanctum protects against the attacks mentioned in § 2, which target the page fault address and cache timing side-channels.

The proof that Sanctum foils page fault attacks is centered around the claims that each enclave’s page fault handler and page tables and page fault handler are isolated from all other software entities on the computer. First, all the page faults inside an enclave’s EVRANGE are reported to the enclave’s fault handler, so the OS cannot observe the virtual addresses associated with the faults. Second, page table isolation implies that the OS cannot access an enclave’s page tables and read the access and dirty bits to learn memory access patterns.

Page table isolation is a direct consequence of the claim that Sanctum correctly protects enclaves against direct attacks, which was covered above. Each enclave’s page tables are stored in DRAM regions allocated to the enclave, so no software outside the enclave can access these page tables.

The proof behind Sanctum’s cache isolation is straightforward but tedious, as there are many aspects involved. We start by peeling off the easier cases, and tackle the most difficult step of the proof at the end of the section. Our design assumes the presence of both per-core caches and a shared LLC, and each cache type requires a separate correctness argument. Per-core cache isolation is achieved simply by flushing per-core caches at every transition between enclave and non-enclave mode. To prove the correctness of LLC isolation, we first show that enclaves do not share LLC lines with outside software, and then we show that the OS cannot indirectly reach into an enclave’s LLC lines via the security monitor.

Showing that enclaves do not share LLC lines with outside software can be accomplished by proving a stronger invariant that states at all times, any LLC line that can potentially cache a location in an enclave’s memory cannot cache any location outside that enclave’s memory. In steady state, this follows directly from the LLC isolation scheme in § 5.1, because the security monitor guarantees that each DRAM region is assigned to exactly one enclave or to the OS.

Last, we focus on the security monitor, because it is the only piece of software outside an enclave that can access the enclave’s DRAM regions. In order to claim that an enclave’s LLC lines are isolated from outside software, we must prove that the OS cannot use the security monitor’s API to indirectly modify the state of the enclave’s LLC lines. This proof is accomplished by considering each function exposed by the monitor API, as well as the monitor’s hardware fault handler. The latter is considered to be under OS control because in a worst case

scenario, a malicious OS could program peripherals to cause interrupts as needed to mount a cache timing attack.

7.3 Operating System Protection

Sanctum protects the operating system from direct attacks against malicious enclaves, but does not protect it against subtle attacks that take advantage of side-channels. Our design assumes that software developers will transition all sensitive software into enclaves, which are protected even if the OS is compromised. At the same time, a honest OS can potentially take advantage of Sanctum’s DRAM regions to isolate mutually mistrusting processes.

Proving that a malicious enclave cannot attack the host computer’s operating system is accomplished by first proving that the security monitor’s APIs that start executing enclave code always place the core in unprivileged mode, and then proving that the enclave can only access OS memory using the OS-provided page tables. The first claim can be proven by inspecting the security monitor’s code. The second claim follows from the correctness proof of the circuits in § 5.2 and § 5.3. Specifically, each enclave can only access memory either via its own page tables or the OS page tables, and the enclave’s page tables cannot point into the DRAM regions owned by the OS.

These two claims effectively show that Sanctum enclaves run with the privileges of their host application. This parallels SGX, so all arguments about OS security in [13] apply to Sanctum as well. Specifically, malicious enclaves cannot DoS the OS, and can be contained using the mechanisms that currently guard against malicious user software.

7.4 Security Monitor Protection

The security monitor is in Sanctum’s TCB, so the system’s security depends on the monitor’s ability to preserve its integrity and protect its secrets from attackers. The monitor does not use address translation, so it is not exposed to any attacks via page tables. The monitor also does not protect itself from cache timing attacks, and instead avoids making any memory accesses that would reveal sensitive information.

Proving that the monitor is protected from direct attacks from a malicious OS or enclave can be accomplished in a few steps. First, we invoke the proof that the circuits in § 5.2 and § 5.3, are correct. Second, we must prove that the security monitor configures Sanctum’s extended page walker registers correctly. Third, we must prove that the DRAM regions that contain monitor code or data are always allocated to the OS.

Since the monitor is exposed to cache timing attacks from the OS, Sanctum’s security guarantees rely on proofs that the attacks would not yield any information that the OS does not already have. Fortunately, most of the secu-

riety monitor implementation consists of acknowledging and verifying the OS’ resource allocation decisions. The main piece of private information held by the security monitor is the attestation key. We can be assured that the monitor does not leak this key, as long as we can prove that the monitor implementation only accesses the key when it is provided to the signing enclave (§ 6.1.2), that the key is provided via a data-independent memory copy operation, such as `memcpy`, and that the attestation key is only disclosed to the signing enclave.

7.5 The Security of Software Attestation

The security of Sanctum’s software attestation scheme depends on the correctness of the measurement root and the security monitor. `mroot`’s sole purpose is to set up the attestation chain, so the attestation’s security requires the correctness of the entire `mroot` code. The monitor’s enclave measurement code also plays an essential role in the attestation process, because it establishes the identity of the attested enclaves, and is also used to distinguish between the signing enclave and other enclaves. Sanctum’s attestation also relies on mailboxes, which are used to securely transmit attestation data from the attested enclave to the signing enclave.

8 Performance Evaluation

While we propose a high-level set of hardware and software to implement Sanctum, we focus our evaluation on the concrete example of a 4-core RISC-V system generated by Rocket Chip [29]. Sanctum conveniently isolates concurrent workloads from one another, so we can examine its overhead via individual applications on a single core, discounting the effect of other running software.

8.1 Experiment Design

We use a Rocket-Chip generator modified to model Sanctum’s additional hardware (§ 5) to generate a 4-core 64-bit RISC-V CPU. Using a cycle-accurate simulator for this machine, coupled with a custom Sanctum cache hierarchy simulator, we compute the program completion time for each benchmark, in cycles, for a variety of DRAM region allocations. The Rocket chip has an in-order single issue pipeline, and does not make forward progress on a TLB or cache miss, which allows us to accurately model a variety of DRAM region allocations efficiently.

We use a vanilla Rocket-Chip as an insecure baseline, against which we compute Sanctum’s overheads. To produce the analysis in this section, we simulated over 250 billion instructions against the insecure baseline, and over 275 billion instructions against the Sanctum simulator. We compute the completion time for various enclave configurations from the simulator’s detailed event log.

Our cache hierarchy follows Intel’s Skylake [23] server models, with 32KB 8-way set associative L1 data and instruction caches, 256KB 8-way L2, and an 8MB 16-way LLC partitioned into core-local slices. Our cache hit and miss latencies follow the Skylake caches. We use a simple model for DRAM accesses and assume unlimited DRAM bandwidth, and a fixed cycle latency for each DRAM access. We also omit an evaluation of the on-chip network and cache coherence overhead, as we do not make any changes that impact any of these subsystems.

Using the hardware model above, we benchmark the integer subset of SPECINT 2006 [3] benchmarks (unmodified), specifically `perlbench`, `bzip2`, `gcc`, `mcf`, `gobmk`, `hmmer`, `sjeng`, `libquantum`, `h264ref`, `omnetpp`, and `astar_base`. This is a mix of memory and compute-bound long-running workloads with diverse locality.

We simulate a machine with 4GB of memory that is divided into 64 DRAM regions by Sanctum’s cache address indexing scheme. Scheduling each benchmark on Core 0, we run it to completion, while the other cores are idling. While we do model its overheads, we choose not to simulate a complete Linux kernel, as doing so would invite a large space of parameters of additional complexity. To this end, we modify the RISC-V proto kernel [48] to provide the few services used by our benchmarks (such as filesystem io), while accounting for the expected overhead of each system call.

8.2 Cost of Added Hardware

Sanctum’s hardware changes add relatively few gates to the Rocket chip, but do increase its area and power consumption. Like SGX, we avoid modifying the core’s critical path: while our addition to the page walker (as analyzed in the next section) may increase the latency of TLB misses, it does not increase the Rocket core’s clock cycle, which is competitive with an ARM Cortex-A5 [29].

As illustrated at the gate level in Figures 8 and 9, we estimate Sanctum to add to Rocket hardware 500 (+0.78%) gates and 700 (+1.9%) flip-flops per core. Precisely, 50 gates for cache index calculation, 1000 gates and 700 flip-flops for the extra address page walker configuration, and 400 gates for the page table entry transformations. DMA filtering requires 600 gates (+0.8%) and 128 flip-flops (+1.8%) in the uncore. We do not make any changes to the LLC, and exclude it from the percentages above (the LLC generally accounts for half of chip area).

8.3 Added Page Walker Latency

Sanctum’s page table entry transformation logic is described in § 5.3, and we expect it can be combined with the page walker FSM logic within a single clock cycle.

Nevertheless, in the worst case, the transformation logic would add a pipeline stage between the L1 data

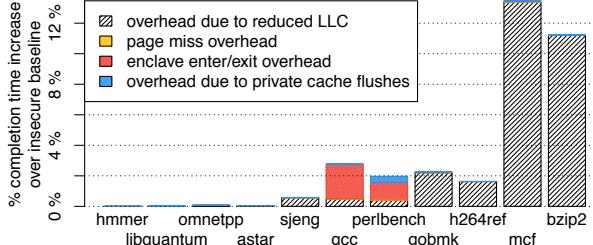


Figure 18: Detail of enclave overhead with a DRAM region allocation of 1/4 of LLC sets.

cache and the page walker. This logic is small and combinational, and significantly simpler than the ALU in the core’s execute stage. In this case, every memory fetch issued by the page walker would experience a 1-cycle latency, which adds 3 cycles of latency to each TLB miss.

The overheads due to additional cycles of TLB miss latency are negligible, as quantified in Figure 18 for SPECINT benchmarks. All TLB-related overheads contribute less than 0.01% slowdown relative to completion time of the insecure baseline. This overhead is insignificant relative to the overheads of cache isolation: TLB misses are infrequent and relatively expensive, several additional cycles makes little difference.

8.4 Security Monitor Overhead

Invoking Sanctum’s security monitor to load code into an enclave adds a one-time setup cost to each isolated process, relative to running code without Sanctum’s enclave. This overhead is amortized by the duration of the computation, so we discount it for long-running workloads.

Entering and exiting enclaves is more expensive than hardware context switches: the security monitor must flush TLBs and L1 caches to avoid leaking private information. Given an estimated cycle cost of each system call in a Sanctum enclave, and in an insecure baseline, we show the modest overheads due to enclave context switches in Figure 18. Moreover, a sensible OS is expected to minimize the number of context switches by allocating some cores to an enclave and allowing them to execute to completion. We therefore also consider this overhead to be negligible for long-running computations.

8.5 Overhead of DRAM Region Isolation

The crux of Sanctum’s strong isolation is caching DRAM regions in distinct sets. When the OS assigns DRAM regions to an enclave, it confines it to a part of the LLC. An enclaved thread effectively runs on a machine with fewer LLC sets, impacting its performance. Note, however, that Sanctum does not partition private caches, so a thread can utilize its core’s entire L1/L2 caches and TLB.

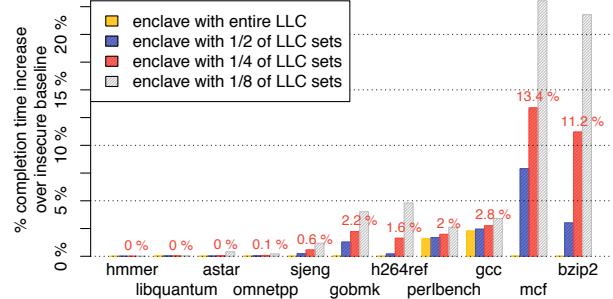


Figure 19: Overhead of enclaves of various size relative to an ideal insecure baseline.

Figure 19 shows the completion times of the SPECINT workloads, each normalized to the completion time of the same benchmark running on an ideal insecure OS that allocates the entire LLC to the benchmark. Sanctum excels at isolating compute-bound workloads operating on sensitive data. SPECINT’s large, multi-phase workloads heavily exercise the entire memory hierarchy, and therefore paint an accurate picture of a worst case for our system. *mcf*, in particular, is very sensitive to the available LLC size, so it incurs noticeable overheads when being confined to a small subset of the LLC. Figure 18 further underlines that the majority of Sanctum’s enclave overheads stem from a reduction in available LLC sets.

We consider *mcf*’s 23% decrease in performance when limited to 1/8th of the LLC to be a very pessimistic view of our system’s performance, as it explores the case where the enclave uses a quarter of CPU power (a core), but 1/8th of the LLC. For a reasonable allocation of 1/4 of DRAM regions (in a 4-core system), enclaves add under 3% overhead to most memory-bound benchmarks (with the exception of *mcf* and *bzip*, which rely on a very large LLC), and do not encumber compute-bound workloads.

9 Conclusion

Sanctum shows that strong provable isolation of concurrent software modules can be achieved with low overhead. This approach provides strong security guarantees against an insidious software threat model including cache timing and memory access pattern attacks. With this work, we hope to enable a shift in discourse in secure hardware architecture away from plugging specific security holes to a principled approach to eliminating attack surfaces.

Acknowledgements: Funding for this research was partially provided by the National Science Foundation under contract number CNS-1413920.

References

- [1] Linux kernel: CVE security vulnerabilities, versions and detailed reports. <http://www.cvedetails.com/>

- [product/47/Linux-Linux-Kernel.html?vendor_id=33](http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33), 2014. [Online; accessed 27-April-2015].
- [2] XEN: CVE security vulnerabilities, versions and detailed reports. http://www.cvedetails.com/product/23463/XEN-XEN.html?vendor_id=6276, 2014. [Online; accessed 27-April-2015].
- [3] SPEC CPU 2006. Tech. rep., Standard Performance Evaluation Corporation, May 2015.
- [4] Xen project software overview. http://wiki.xen.org/wiki/Xen_Project_Software_Overview, 2015. [Online; accessed 27-April-2015].
- [5] ANATI, I., GUERON, S., JOHNSON, S. P., AND SCARLATA, V. R. Innovative technology for CPU based attestation and sealing. In *HASP* (2013).
- [6] ANTHONY, S. Who actually develops linux? the answer might surprise you. <http://www.extremetech.com/computing/175919-who-actually-develops-linux>, 2014. [Online; accessed 27-April-2015].
- [7] BANESCU, S. Cache timing attacks. [Online; accessed 26-January-2014].
- [8] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems-CHES 2006*. Springer, 2006, pp. 201–215.
- [9] BRUMLEY, B. B., AND TUVERI, N. Remote timing attacks are still practical. In *Computer Security—ESORICS*. Springer, 2011.
- [10] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. *Computer Networks* (2005).
- [11] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Asia-Pacific Workshop on Systems* (2011), ACM.
- [12] CHHABRA, S., ROGERS, B., SOLIHIN, Y., AND PRVULOVIC, M. SecureME: a hardware-software approach to full system security. In *international conference on Supercomputing (ICS)* (2011), ACM.
- [13] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, Feb 2016.
- [14] DAVENPORT, S. SGX: the good, the bad and the downright ugly. *Virus Bulletin* (2014).
- [15] DOMNITSER, L., JALEEL, A., LOEW, J., ABU-GHAZALEH, N., AND PONOMAREV, D. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *Transactions on Architecture and Code Optimization (TACO)* (2012).
- [16] DUFLOT, L., ETIEMBLE, D., AND GRUMELARD, O. Using CPU system management mode to circumvent operating system security functions. *CanSecWest/core06* (2006).
- [17] DUNN, A., HOFMANN, O., WATERS, B., AND WITCHEL, E. Cloaking malware with the trusted platform module. In *USENIX Security Symposium* (2011).
- [18] EMBLETON, S., SPARKS, S., AND ZOU, C. C. SMM rootkit: a new breed of os independent malware. *Security and Communication Networks* (2010).
- [19] EVTYUSHKIN, D., ELWELL, J., OZSOY, M., PONOMAREV, D., ABU GHAZALEH, N., AND RILEY, R. Iso-X: A flexible architecture for hardware-managed isolated execution. In *Microarchitecture (MICRO)* (2014), IEEE.
- [20] FLETCHER, C. W., DIJK, M. V., AND DEVADAS, S. A secure processor architecture for encrypted computation on untrusted programs. In *Workshop on Scalable Trusted Computing* (2012), ACM.
- [21] GOLDRICH, O. Towards a theory of software protection and simulation by oblivious RAMs. In *Theory of Computing* (1987), ACM.
- [22] GRAWROCK, D. *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.
- [23] INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Sep 2014. Reference no. 248966-030.
- [24] KESSLER, R. E., AND HILL, M. D. Page placement algorithms for large real-indexed caches. *Transactions on Computer Systems (TOCS)* (1992).
- [25] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA* (2014), IEEE Press.
- [26] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. seL4: Formal verification of an OS kernel. In *SIGOPS symposium on Operating systems principles* (2009), ACM.
- [27] KOCHER, P. C. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *Advances in Cryptology (CRYPTO)* (1996), Springer.
- [28] KONG, J., ACIICMEZ, O., SEIFERT, J.-P., AND ZHOU, H. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *workshop on Computer security architectures* (2008), ACM.
- [29] LEE, Y., WATERMAN, A., AVIZENIS, R., COOK, H., SUN, C., STOJANOVIC, V., AND ASANOVIC, K. A 45nm 1.3 ghz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *European Solid State Circuits Conference (ESSCIRC)* (2014), IEEE.

- [30] LIE, D., THEKKATH, C., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. Architectural support for copy and tamper resistant software. *SIGPLAN Notices* (2000).
- [31] LIN, J., LU, Q., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA* (2008), IEEE.
- [32] LIU, C., HARRIS, A., MAAS, M., HICKS, M., TIWARI, M., AND SHI, E. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS* (2015).
- [33] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA* (Mar 2016).
- [34] LIU, F., AND LEE, R. B. Random fill cache architecture. In *Microarchitecture (MICRO)* (2014), IEEE.
- [35] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Security and Privacy* (2015), IEEE.
- [36] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. *HASP* (2013).
- [37] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox – practical cache attacks in javascript. *arXiv preprint arXiv:1502.07373* (2015).
- [38] RUTKOWSKA, J. Thoughts on intel’s upcoming software guard extensions (part 2). *Invisible Things Lab* (2013).
- [39] RUTKOWSKA, J., AND WOJTCZUK, R. Preventing and detecting xen hypervisor subversions. *Blackhat Briefings USA* (2008).
- [40] SANCHEZ, D., AND KOZYRAKIS, C. The ZCache: Decoupling ways and associativity. In *Microarchitecture (MICRO)* (2010), IEEE.
- [41] SANCHEZ, D., AND KOZYRAKIS, C. Vantage: scalable and efficient fine-grain cache partitioning. In *SIGARCH Computer Architecture News* (2011), ACM.
- [42] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM rowhammer bug to gain kernel privileges. <http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html>, Mar 2015. [Online; accessed 9-March-2015].
- [43] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path oram: An extremely simple oblivious ram protocol. In *SIGSAC Computer & communications security* (2013), ACM.
- [44] SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *international conference on Supercomputing (ICS)* (2003), ACM.
- [45] TAYLOR, G., DAVIES, P., AND FARMWALD, M. The TLB slice - a low-cost high-speed address translation mechanism. *SIGARCH Computer Architecture News* (1990).
- [46] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *International Symposium on Computer Architecture (ISCA)* (2007).
- [47] WATERMAN, A., LEE, Y., AVIZIENIS, R., PATTERSON, D. A., AND ASANOVIC, K. The RISC-V instruction set manual volume II: Privileged architecture version 1.7. Tech. Rep. UCB/EECS-2015-49, EECS Department, University of California, Berkeley, May 2015.
- [48] WATERMAN, A., LEE, Y., AND CELIO, CHRISTOPHER, E. A. RISC-V proxy kernel and boot loader. Tech. rep., EECS Department, University of California, Berkeley, May 2015.
- [49] WATERMAN, A., LEE, Y., PATTERSON, D. A., AND ASANOVIC, K. The RISC-V instruction set manual, volume i: User-level ISA, version 2.0. Tech. Rep. UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.
- [50] WECHEROWSKI, F. A real SMM rootkit: Reversing and hooking BIOS SMI handlers. *Phrack Magazine* (2009).
- [51] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking intel trusted execution technology. *Black Hat DC* (2009).
- [52] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking SMM memory via intel CPU cache poisoning. *Invisible Things Lab* (2009).
- [53] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking intel TXT via SINIT code execution hijacking, 2011.
- [54] WOJTCZUK, R., RUTKOWSKA, J., AND TERESHKIN, A. Another way to circumvent intel® trusted execution technology. *Invisible Things Lab* (2009).
- [55] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Oakland* (May 2015), IEEE.
- [56] YAROM, Y., AND FALKNER, K. E. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive* (2013).
- [57] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy* (2009), IEEE.

Ariadne: A Minimal Approach to State Continuity

Raoul Strackx

*iMinds-DistriNet, KU Leuven,
3001 Leuven, Belgium
raoul.strackx@cs.kuleuven.be*

Frank Piessens

*iMinds-DistriNet, KU Leuven,
3001 Leuven, Belgium
frank.piessens@cs.kuleuven.be*

Abstract

Protected-module architectures such as Intel SGX provide strong isolation guarantees to sensitive parts of applications while the system is up and running. Unfortunately systems in practice crash, go down for reboots or lose power at unexpected moments in time. To deal with such events, additional security measures need to be taken to guarantee that stateful modules will either recover their state from the last stored state, or fail-stop on detection of tampering with that state. More specifically, protected-module architectures need to provide a security primitive that guarantees that (1) attackers cannot present a stale state as being fresh (i.e. rollback protection), (2) once a module accepted a specific input, it will continue execution on that input or never advance, and (3) an unexpected loss of power must never leave the system in a state from which it can never resume execution (i.e. liveness guarantee).

We propose Ariadne, a solution to the state-continuity problem that achieves the theoretical lower limit of requiring only a single bit flip of non-volatile memory per state update. Ariadne can be easily adapted to the platform at hand. In low-end devices where non-volatile memory may wear out quickly and the bill of materials (BOM) needs to be minimized, Ariadne can take optimal use of non-volatile memory. On SGX-enabled processors, Ariadne can be readily deployed to protect stateful modules (e.g., as used by Haven and VC³).

1 Introduction

Computing devices have become ever more diverse, ranging from cloud computing platforms and super computers to embedded systems used in Internet of Things (IoT) applications. The familiar multi-level approach to security where a more privileged layer has full control over software running on top, is ill-suited for many of these applications; clients of cloud providers may

fear rogue employees or government subpoenas targeting their provider that may reside in a different country [7]. On embedded devices paging and privilege layers may be too power/energy expensive to be applied.

Protected-module architectures (PMAs) take another, non-hierarchical approach. After almost a decade of research [5, 6, 8, 9, 11–13, 22, 27, 28, 30, 31, 41, 45, 48, 49], two key primitives have emerged: isolation and key derivation. The *isolation* mechanism ensures that a protected module is completely isolated from any other piece of code running on the system, including other protected modules. Only when the instruction pointer points to a memory location within the module, can a module’s memory regions be accessed. All other attempts from different locations are blocked by the architecture. Only a module’s entry points are an exception and can be accessed from any location. Once an entry point is called, the instruction pointer points within the module’s memory region and its secrets can be accessed.

A *key derivation mechanism* provides a unique, unforgeable key for each protected module. It is usually derived from a platform specific key – that can only be accessed directly by the platform itself – and the measurement of the module when it was created. This implies that only identical modules can derive the same cryptographic key. An attacker modifying the module before it was properly isolated, will cause a variation in the module’s measurement and eventually in the cryptographic key that was derived. This makes it ideal to seal data to a specific module. Data can be integrity and confidentiality protected by the derived key, and stored on disk. As only the identical module can derive the same key, the stored sensitive data cannot be accessed by an attacker.

Related work showed that these minimal requirements are small enough to be implemented directly in hardware [31], even for embedded devices [8, 12, 22, 30]. With the arrival of Intel Skylake CPUs in August 2015, equipped with Intel Software Guard eXtensions (SGX) [3, 18, 29], PMAs are now available on commod-

ity devices.

In parallel, research was conducted on how the security properties provided by PMAs can be leveraged to provide provable security guarantees. Agten et al. [2] and others in subsequent work [1, 33, 34] showed that by adding limited security checks at runtime, fully-abstract compilation can be guaranteed; all security properties that hold at source-code level, can be guaranteed at machine-code level too. This makes reasoning about security properties much easier.

Unfortunately an important security requirement has received little attention. Many security properties only hold while the system is up and running. In practice machines crash, go down for reboots and lose power at unexpected moments in time. To account for such events, stateful protected modules must securely store their state. Parno et al. [32] showed that this is a non-trivial task. Sealing a module’s state before it is handed over to the untrusted operating system and written to disk, is insufficient. Additional security measures need to be taken to ensure that: (1) a protected module’s state can never be rolled back to a previous, stale state, (2) once a module accepted input, it must either (eventually) finish its execution or never advance at all and (3) unexpected loss of power at any moment in time should never result in a system that cannot be resumed after reboot.

State continuity solutions must take the platform specifications and use case at hand into account. Existing solutions [32, 46] rely on an uninterruptible power source (UPS) or risk wearing out non-volatile memory. Such approaches are acceptable in higher-end applications (e.g., in a cloud setting [7, 40]). On other platforms [35, 36] a UPS may not be available on commercial off-the-shelf (COTS) devices and may lead to significant increases in the bill-of-materials (BOM). In such cases wear and tear on non-volatile memory must be minimized to increase longevity of the device. We present a proven-secure solution to state-continuity and show that it can be applied on a large range of platforms.

More specifically, we make following contributions:

- We present Ariadne, a solution to the state-continuity problem that achieves the mathematical lower-bound of only a single bit flip per state update.
- We show that Ariadne can be easily adapted to reduce wear on EEPROM/NAND flash memory.
- We demonstrate that Ariadne can be applied immediately on the SGX/ME platform, without any hardware modifications. This is particularly important as SGX modules (called enclaves) are destroyed when the system is suspended or hibernated. In addition we compare the use of the Intel Manage-

ment Engine (ME) to the TPM chip to store freshness information and show that no clear winner exists; much depends on the use case and the available hardware.

The remainder of this paper is structured as follows. In the next section we discuss our attack model and the security properties we need to provide in detail. Section 3 builds upon related work and shows that the state-continuity problem can be easily reduced to that of state-continuous storage. Ariadne’s algorithm and its optimizations are discussed in Section 4. We evaluate its security in Section 5. Two possible implementations are discussed and evaluated in Section 6. Finally we discuss related work and conclude in Sections 7 and 8.

2 Problem Definition

2.1 Attack Model

Our goal is to provide state-continuity support to protected-module architectures, without (significantly) increasing their attack surface. As such, we assume the following. First, an attacker is able to compromise the complete (untrusted) software stack. As untrusted operating system services are used to store and retrieve module states, this implies that these states can be replayed.

Second, an attacker is able to halt execution at any moment in time because she has complete control over the system’s power supply, or because she can launch other attacks leading to similar results. Especially Intel SGX is vulnerable to such attacks. To protect the system from malicious or badly behaving enclaves, execution control is returned to the kernel whenever an interrupt occurs while an enclave is executing. Regardless of how such attacks are executed, we will refer to them as “power-interruption attacks”.

2.2 Security Properties

In order to build a secure and reliable system, it is paramount that we are able to provide three security properties. First, we need to provide rollback prevention; an attacker must not be able to provide a stale state of a module and have it accepted as being fresh. Especially in DRM/ERM contexts such security guarantees are important. Consider as an example a document that should only be printed a limited number of times. Such limitations can be guaranteed easily by first checking and decrementing a monotonic counter before the document is printed.

Second, a module’s execution needs to be continuous. Once a module accepted input, it needs to eventually finish its execution based on that input and output all com-

puted results, or it must never advance at all. This property is related to rollback-prevention, but is much stricter. Rollback-prevention may guarantee for example that an X509 certificate authority (CA) does not provide two different certificates with the same serial number. But in practice it is also important that every certificate is accounted for; for every serial number the CA should be able to prove which (if any) unique certificate it signed. Failure to do so may break trust [17] that it did not provide rogue certificates.

Third, we must also be able to guarantee liveness of the system. Unexpected crashes or loss of power at *any* moment in time, must not result in a system that will never be able to recover. Note that this is not an availability guarantee. We only consider interrupts in execution that may occur even if the system is not under attack. We cannot guarantee availability: a kernel-level attacker can easily prevent a system from ever resuming its previous state by erasing the fresh state from disk, enter an endless crash-reboot cycle, or erasing the system’s boot image.

3 Background: State-Continuous Storage is Sufficient

Related work [32, 46] already showed that the state-continuity problem can be reduced to that of state-continuous storage. We take the same approach. We first introduce `libariadne`, a library providing state-continuous storage in Section 3.1. Sections 3.2 and 3.3 discuss how this library should be used and introduce a running example.

3.1 libariadne’s Interface

We provide programmers with the `libariadne` library that can be linked with a protected module. It provides an interface of three functions.

The `void store_state(Blob *blob, String f_format)` function stores the provided data in `blob` in a file on disk. To enable easy recovery of the fresh state, we require that `f_format` is a format string which includes an integer conversion specifier (i.e., “%i”). The library will internally replace the conversion specifier with the value of a monotonic counter. Note that this is only for practical reasons, an attacker changing the filename may prevent the module from ever being resumed, but it will never result in a rollback attack.

The `Blob *retrieve_state(String f_format)` function will attempt to read a file with a matching filename, verify its integrity and freshness and return its content. When this verification step fails for any reason `NULL` is returned.

In case an attacker deletes the fresh data from disk – or simply when the hard drive got damaged and

needed replacement – the fresh data is permanently lost. The `void purge_state(Blob *init, String format)` function can resolve the situation by allowing the programmer to specify an initial, public state of the module. As this results in a loss of any previously stored sensitive data, this operation does not violate state continuity.

3.2 Security Considerations of the libariadne Library

With `libariadne` providing state-continuous storage, protected modules writers can easily guarantee rollback prevention and continuous execution [32, 46] by ensuring that modules adhere to two principles:

Requirement 1: Store Input Before Processing *Before* processing any input, protected modules must store their current state, with the received input and called entry point.¹

Requirement 2: Only Deterministic Protected Modules Any source of non-determinism (e.g., `rdrand` instructions) needs to be considered input, and thus following Requirement 1 stored before being used.

These requirements ensure that when a protected-module is interrupted during execution (e.g., due to a power failure), it will restart the computation based on the *same* input when the module is resumed. Since modules are deterministic, it will either reach an identical state as when power was lost, or its execution is interrupted again before it reaches that state.

3.3 Running Example: A PIN-Protected Secret

Consider a module that protects access to a secret. Only when a user presents a valid PIN, will the secret be returned. To mitigate brute-force attacks, we need to be able to guarantee that a user/attacker can make at most three failed attempts before being locked out indefinitely.

The implementation of the module is presented in listing 1. Whenever the module is loaded in memory, the `on_load` function is called implicitly, and an attempt is made to retrieve the last stored state (line 8). If `libariadne`’s `retrieve_state` function finds a freshly stored file, the module’s state is restored and execution of the last called entry point is restarted. Eventually the module will end up in the same state as when the module was interrupted.

If on the other hand no matching file can be found, it is corrupted, or it is stale, `retrieve_state` will return

¹Alternatively we could have opted to state-continuously store the state only right before the module returns output. Unfortunately this is hard in practice as any sources of output need to be considered (e.g., calls to unprotected memory, and timing and page-fault channels [59]).

```

1 #include <libariadne/interface.h>
2
3 static int tries_left;
4 static String pin;
5 static String secret;
6
7 void on_load( void ) {
8     Blob *blob = retrieve_state( "state_%i.pkg" );
9
10    if ( blob != NULL ){
11        // restart computation using state & input in blob
12        ...
13    } else
14        reset();
15 }
16
17 void entry_point reset( void ) {
18     Blob *blob = new Blob( &reset || tries_left || pin ||
19         secret );
20     purge_state( blob, "state_%i.pkg" );
21     pin = "0000";
22     secret = "publicly-known secret";
23     tries_left = 3;
24 }
25
26 String entry_point get_secret( String p ) {
27     Blob *blob = new Blob( &get_secret || p || tries_left ||
28         pin || secret );
29     store_state( blob, "state_%i.pkg" );
30
31     if ( tries_left <= 0 )
32         return "Locked out";
33
34     if ( pin == p ) {
35         tries_left = 3;
36         return secret;
37     } else {
38         --tries_left;
39         return "Incorrect PIN";
40     }
41 }
42 bool entry_point set_pin( String p_old, String p_new ){...}
43 bool entry_point set_secret( String p, String s_new ){...}

```

Listing 1: A running example: A PIN-protected secret. The `||` operator is used to denote concatenation.

NULL. It is up to the module writers to handle such situations. In this case the module is `reset` to a known good initial state (line 14), at the cost of losing the protected secret indefinitely. The same `reset` entry point can be called when the user exhausted her 3 access attempts. Note that the `reset` function stores the previous `tries_left`, `pin` and `secret` within the created blob. While this is not required given that the function will al-

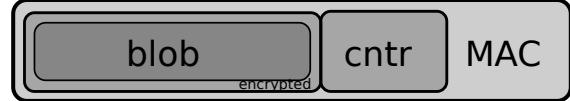


Figure 1: Layout of a package. A module’s state and input is confidentiality and integrity protected. The enclosed `cntr` value enables Ariadne to determine freshness.

ways set these variables to a known-good value, it does *not* enable an avenue of attack; Ariadne will always ensure that the `reset` function will be called upon recovery.

When `on_load` successfully retrieved and resumed the fresh state, the user can attempt to access the secret by calling the `get_secret` entry point and providing it with a PIN. When she hasn’t exhausted her number of guesses yet, the provided PIN `p` is verified (line 32). When the correct PIN is provided, the `tries_left` variable is reset and the secret is returned. Otherwise `tries_left` is decremented and an error string is returned. The entry points `set_pin` and `set_secret` – offering the obvious functionality to the user – use identical security measures to protect against brute-force attacks.

4 Ariadne

We describe Ariadne in three steps. In Section 4.1 we provide a scheme for state-continuous storage based on a monotonic counter. In Sections 4.2 and 4.3 we propose alternative counter encodings that will lead to a minimal approach to state continuity.

We will only focus on state-continuity guarantees for a single module. Related work [32, 46] showed that support for an unlimited amount of modules can be added easily by (1) using a single state-continuous module to provide secure, state-continuous storage for other modules and (2) inter-module communication. For completeness, we elaborate in Appendix B.

4.1 State-Continuity based on a Monotonic Counter

Guaranteeing state continuity based only on a single monotonic counter to keep track of the fresh state, is a hard problem. Developers of protected modules may be tempted to re-use solutions borrowed from anti-replay security measures, but these are flawed.

Strawman Solutions Generally two approaches are considered. In one approach the monotonic counter is incremented first. Afterwards, the state of the module is confidentiality protected, appended with the counter

value and the whole is integrity protected. Figure 1 displays the resulting *package* graphically. When the system crashes, only the package with an encapsulated counter value equal to the monotonic counter is accepted as being fresh. This approach has the obvious problem that a crash before the new package was written to disk, prevents the system from recovering; liveness cannot be guaranteed.

Alternatively, first storing the package with the next counter value *before* the monotonic counter is incremented also fails. Repeated crashes before the monotonic counter was incremented, enables the creation of multiple packages with the same counter values but different user input. This enables dictionary-style attacks and thus breaks rollback prevention and continuity guarantees. We elaborate on both attacks in Appendix A.

Key Observations Ariadne relies on two important observations. First, the isolation guarantees of protected module architectures ensure that an attacker cannot jump within the middle of a module. During execution we are thus able to assemble guarantees that may not hold when power is lost unexpectedly.

Second, an attacker is extremely restricted in the valid packages she can get access to. The MAC included in the package prevents her from crafting her own packages, or modifying existing ones. To ensure liveness, we need to write new packages to disk before incrementing the monotonic counter. An attacker can abuse this behavior by crashing the system before the counter is incremented. But this implies that at any moment in time, she has at most access to packages with an enclosed counter value smaller than one increment of the monotonic counter.

Ariadne’s key insight is that during recovery from a crash, we need to store the fresh package and increment the monotonic counter *twice* before the encapsulated state is resumed.

A Secure Solution Let’s re-use the PIN-protected module to describe our solution. Assume that the module is up and running. When a user requests access to the secret by calling `get_secret` (listing 1, line 25), the input and state of the module is placed in a new blob and the `store_state` function is called. Listing 2 displays its implementation. To ensure liveness, a new package is created with the next counter value and stored on disk. Finally the monotonic counter is incremented.

When the module needs to be re-loaded in memory, its `on_load` function is called implicitly (listing 1, line 7) and the `retrieve_state` library function is called to retrieve its fresh state. A package is read from disk and only accepted as being fresh *iff* its MAC value is verified successfully and its enclosed counter value matches with the value of the monotonic counter (listing 2, line 13-19).

```

1 #include <libariadne/interface.h>
2
3 void store_state( Blob *blob, String f_format ){
4     Package *pkg = create_pkg( blob, hwcntr.value() + 1 );
5     hdd.write( pkg, f_format, hwcntr.value() + 1 );
6     hwcntr.inc();
7 }
8
9 Blob *retrieve_state( String f_format ){
10    Package *pkg;
11    Blob *blob;
12
13    pkg = hdd.read( f_format, hwcntr.value() );
14
15    if ( pkg == NULL || !auth( pkg, get_mac_key() ) )
16        return NULL;
17
18    if ( pkg->cntr != hwcntr.value() )
19        return NULL;
20
21    blob = decrypt( pkg, get_enc_key() );
22
23    // ∀pkg ∈ hdd : pkg->counter ≤ hwcntr.value() + 1
24    pkg = create_pkg( blob, hwcntr.value() + 1 );
25    hdd.write( pkg, f_format, hwcntr.value() + 1 );
26    hwcntr.inc();
27
28    // ∀pkg ∈ hdd : pkg->counter ≤ hwcntr.value()
29    pkg = create_pkg( blob, hwcntr.value() + 1 );
30    hdd.write( pkg, f_format, hwcntr.value() + 1 );
31    hwcntr.inc();
32
33    // ∀pkg ∈ hdd : pkg->counter ≤ hwcntr.value()
34    // ∀pkg ∈ hdd : pkg->counter = hwcntr.value() →
35    //                 pkg->contents = blob
36    return blob;
37
38 void purge_state( Blob *init_blob, String f_format ){
39    hwcntr.inc();
40
41    Package *pkg = create_pkg( blob, hwcntr.value() + 1 );
42    hdd.write( pkg, f_format, hwcntr.value() + 1 );
43    hwcntr.inc();
44 }
```

Listing 2: Ariadne: State-Continuity based on a monotonic counter

Now that we determined that the package read from disk is fresh, we can resume the execution of the module. But before doing so, we need to guarantee that no other packages exist with the same counter value. We already observed that an attacker may have packages with an encapsulated counter value of one increment larger than the monotonic counter. Incrementing the monotonic counter twice will thus guarantee that all attacker’s pack-

ages are seen as stale in the future. Only incrementing the counter once is not sufficient. We describe an attack against this scheme in Appendix A.3. To guarantee liveness, new packages are written to disk before the monotonic counter is incremented.

To restart a module from a known-good state, `purge_state` can be called (listing 2, line 38). Similar to the `retrieve_state` function, we need to guarantee that when the function returns, there only exists a single package that will be accepted as being fresh. Hence, in this function too we need to increment the monotonic counter twice. However, since this function can always be restarted, liveness is no longer a concern. We can therefore omit storing a new package before we increment the monotonic counter for the first time (line 39).

Skipping Unprocessed Input A careful reader may have noticed that it is possible for an attacker to force the creation of packages with the same enclosed counter value but with different user input. We will show that this does not break state continuity.

Let’s use the PIN-protected module again as an example. The attempted attack goes as follows: the attacker calls the `get_secret` function and provides a PIN p . After the module assembled a `Blob` structure containing the current state of the module, the provided input and the entry point used (listing 1, line 27), the `store_state` library function is called. There the attacker crashes the system right before the monotonic counter is incremented (listing 2, line 6). Since a new package was already written to disk, she now possesses a package with enclosed the next value of the monotonic counter and the provided PIN p .

The attacker now has two options. Neither will break state continuity. The first option is to resume the system without any interference. As the newly written package is not seen as being fresh (the monotonic counter in non-volatile memory was not yet incremented) PIN guess p will be discarded. As this guess was never compared to the real PIN code, the attacker did not learn any new information and state-continuity remains guaranteed.

Alternatively, the attacker lets the system reboot but crashes the module immediately after the non-volatile counter is incremented for the first time in the `retrieve_state` function (Listing 2, line 26). When the system now recovers again, the package with PIN guess p will be seen as being fresh. As no input was processed by the module after this guess was made, state-continuity is also guaranteed in this case. As input can only be skipped until the module completes its `retrieve_state` function, repeated crashes during `retrieve_state` will also not break state continuity.

4.2 State-Continuity by Flipping Bits

Related work [32, 46] relied on the irreversibility of hash values to keep track of freshness information. In order to avoid that unexpected loss of power while the hash value is being updated may lead to corrupted non-volatile memory, both approaches required a 2-phase commit protocol. We take a different approach. We use the state-continuity approach based on monotonic counters in the previous section, but use a counter encoding that only requires a single bit flip per state update. In the next section we will show that in practical implementations we can guarantee that these operations can be implemented atomically, and we thus avoid a need of a 2-phase commit protocol altogether. As every solution to state-continuity requires at least a single flip to be recorded, we reached a theoretical lower limit.

Balanced Gray codes provides such an encoding, with the additional benefit that every bit is (almost) equally used. Construction of these codes with arbitrary lengths is non-trivial. In 2008 a constructive proof of their existence was presented by Mary Flahive [14] but to the best of our knowledge never implemented. We present a fast algorithm with fixed, limited, memory consumption.

4.2.1 Terminology

Gray codes ensure that two adjacent code words differ in only a single digit. For example:

$$\{0\underline{0}, \underline{0}1, 1\underline{1}, \underline{1}0\} \quad (1)$$

and

$$\{00\underline{0}, \underline{0}01, 01\underline{1}, \underline{0}10, 11\underline{0}, \underline{1}11, 10\underline{1}, \underline{1}00\} \quad (2)$$

are two- and three-digit Gray codes, respectively. Moreover, these encodings are cyclic as the same property applies to the last and first code word. The underlined digits in encoding 1 and 2 are the *transition digits* and show which digit is changed in the next code word. The transition count of a digit is the number of times that digit is used. For example, digit 0 in encoding 2 is used twice.² The collection of these transition counts is called the transition spectrum of the Gray code (i.e., $(2, 2)$ and $(2, 2, 4)$ for encoding 1 and 2 respectively). When all transition counts are equal (t.i. $2^n/n$ with n the length of the Gray code), the encoding is said to be *uniform* or *completely balanced*. Obviously this can only be achieved when $2^n/n$ is an integer. In other cases *cyclic balanced Gray Codes* can be constructed where the difference of any pair of transition counts is at most two:

$$\forall 0 \leq i, j < n : |TC_n(j) - TC_n(i)| \leq 2 \quad (3)$$

²We follow the convention that Gray codes start with digit 0 on the left hand side.

4.2.2 Construction

Let's use the construction of a 5-bit balanced Gray code as a running example. The algorithm is displayed in Figure 2. Balanced Gray Codes of length n are constructed recursively from $n - 2$ bit balanced Gray Codes. The balanced 2 and 3-bit Gray codes of encoding 1 and 2 are used as base cases. Each iteration consists of three steps. In the first step, a $2^{n-2} \times 2^2$ grid is constructed. The rows are annotated with all $n - 2$ digit Gray codes. The number of columns is fixed for any n and columns are annotated with 2-digit codes. A given vertex now represents an n -bit Gray code by concatenating the Gray code of the row and the column. By construction, two (toroidally) adjacent vertices will now represent adjacent Gray codes. Every Hamiltonian cycle found in this grid now represents an encoding, but care needs to be taken to ensure that it is balanced.

In the second step we partition the grid such that the transition counts within each partition can be easily calculated. Partitions are represented as black boxes in Figure 2. Calculation where a new partition needs to be started is discussed in Section 4.2.3. For now note that we will ensure that (1) new partitions will always start at the first and last row and (2) the number of partitions is even.

Finally, the Hamiltonian cycle is constructed starting with Gray code 00000. With the exception of the last two partitions, each partition is traversed in the same way: visit every vertex in the column before moving to the next column. Whenever an edge of a partition is reached, the horizontal/vertical direction is inverted.

Let's take the fourth partition – the first partition containing more than one row – as an example. The partition is started at vertex with Gray code word 01011 when we were in the third column and going from right to left. The graph is continued by first traversing all rows down the current partition. The digit on the edge is the transition digit used to create the next Gray code. When the partition's end is reached (vertex 11011), we move to the next column (vertex 11001) and the rows in the partition are revisited in reversed order. When the start of the partition is reached again (vertex 01001), the last column is selected and each row is again traversed. Finally, the end of the partition is reached (vertex 11000).

The last and second to last partition are constructed differently as shown in Figure 2. We will show that the created notch in the last two partitions ensures that each partition has exactly the same properties.

4.2.3 Computing Partition Sizes

Based on the construction of the Hamiltonian cycle, we can easily derive the transition counts $TC_n(i)$ of each transition digit i of the resulting n -digit Gray code. We

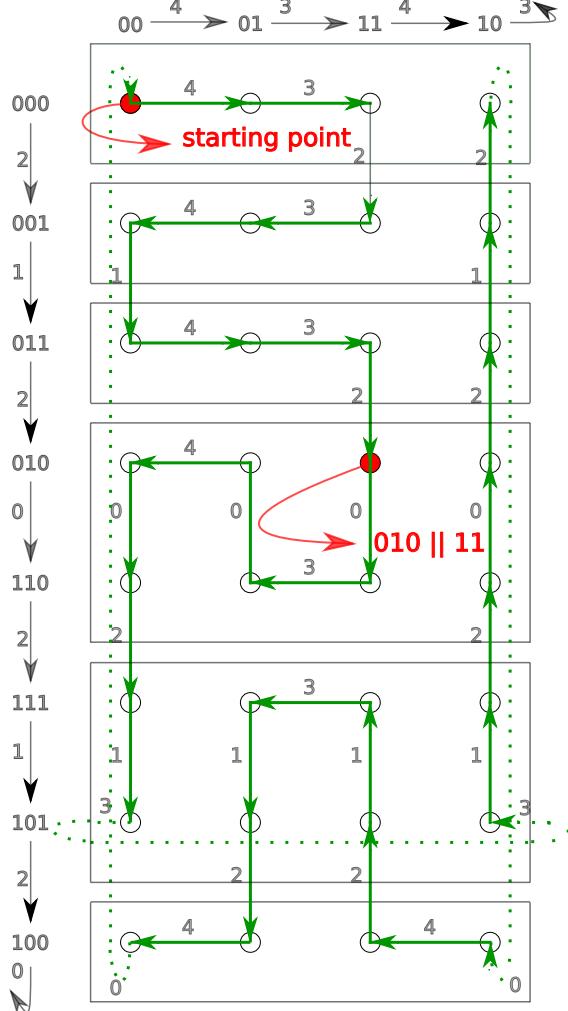


Figure 2: Balanced n -bit Gray Codes can be constructed from $n - 2$ balanced Gray Codes

apply a separate reasoning for digits i smaller than $n - 2$ (vertical arrows) and digits $n - 2$ and $n - 1$ (horizontal arrows).

Within a partition vertical arrows are always used 4 times. When a vertical arrow is used as a *connecting digit* – connecting two partitions – it is used twice. This totals the transition counts for these digits to: $4(TC_{n-2}(i) - m_{n-2}(i)) + 2m_{n-2}(i)$ where $m_{n-2}(i)$ is the multiplicity of digit i ; the number of times transition digit i is used as a connecting digit.

Transition digits $n - 2$ and $n - 1$ are represented by horizontal arrows in Figure 2. For standard partitions each is used once per partition. The second to last partition uses $n - 2$ twice, but $n - 1$ isn't used. In the last partition the opposite happens; $n - 1$ is used twice, but $n - 2$ isn't used. For the entire graph, transition digits $n - 2$ and $n - 1$ are thus used L times, with L the number

of partitions. As each partition is connected to the next with a connecting digit, we know that $L = \sum_{i=0}^{n-2} m_{n-2}(i)$. The transition counts for the generated Gray code is thus:

$$TC_n(i) = \begin{cases} 4TC_{n-2}(i) - 2m_{n-2}(i) & \text{if } i < n-2 \\ \sum_{i=0}^{n-2} m_{n-2}(i) & \text{if } i \geq n-2 \end{cases} \quad (4)$$

Flahive [14] showed that a set of connecting digits can always be found such that a balanced Gray code is constructed.

4.2.4 Generating Balanced Gray Codes

Flahive's construction guarantees that we will generate a balanced Gray code when transition digit i is used as a connecting digit between partitions exactly $m_{n-2}(i)$ times. Unfortunately implementing it directly is not possible for large n ; keeping track of each partition would quickly consume too much memory. Instead we only precompute $m_{n-2}(i)$ for each $0 \leq i < n-2$ and apply a greedy algorithm that will exhaust each m_i as fast as possible.

Let's reuse the construction of a 5-digit balanced Gray code as an example. Solving equation 4 so that a balanced Gray code is constructed (see equation 3), we get $TC_n(i) = (6, 6, 8, 6, 6)$ with $m_{n-2}(i) = (1, 1, 4)$.

In order to find a graph that fulfills these requirements, we keep track of the size of the current partition, at which row and column we are currently at, in which horizontal and vertical direction we are going and for each transition digit how many connecting digits are still available. Starting at vertex 00000 going in a downwards and right direction (see Figure 2), we calculate (recursively) the transition digit going downward (t.i. 2). As transition digit 2 still needs to be used (4 times) as a connecting digit, we decide to stop the current partition immediately, revert the vertical direction taken and switch columns instead.

Similarly, at vertex 01011 we determine that 0 would be used as a transition digit if we do not terminate the partition. While this transition digit was not yet used as a connecting digit and it should be used once, we do not stop the current partition. Indeed, by construction transition digit 0 is always used as a connecting digit between the last and first partition.

It is important to note that the metadata (in the order of KB) used to generate the next Gray codes, can be included in the package written to disk. Close examination of libariadne's implementation shows that in order to determine whether a package is fresh, we only need to compare the package's counter with the hardware monotonic counter (listing 2, line 18). In other words, we only need access to the Gray codes' metadata after we have already determined that the package is fresh.

4.3 Optimizing for Program-Erase Cycles

Being able to provide state-continuity guarantees by only flipping a single bit per state update, isn't just a theoretical result. It enables various additional optimizations. Let's assume that we use EEPROM/NAND flash memory to store the monotonic counter as an example.

EEPROM is 1980s technology that is still used in some TPM chips [4, 43, 44]. For most applications however it has been replaced by flash memory (e.g., Intel's Management Engine (ME) [37]) because it is cheaper to manufacture for bigger memory sizes. This however comes at a cost; while EEPROM is byte accessible, NAND flash memory needs to be addressed in bigger units. Read and write operations are page-based of usually 2KB to 8KB. Erase commands on the hand operate on blocks of 32-128 pages. [55]

Both EEPROM and flash memory have the disadvantage that they age. Every time the memory is written to or erased, high voltages are applied that eventually will damage the device oxide. Eventually memory cells will be in a stuck-at state, or fail to retain their information over longer periods of time. The number of program/erase (P/E) cycles that can reliably be issued, depends heavily on the manufacturing process (density, single/multi-layer cells, etc.), but typically ranges between 5,000 and 500,000 cycles. [42]

Being provided with different commands to write and erase memory gives us an opportunity to optimize our encoding scheme further. Erasing a block will set all memory cells to 1, while subsequent write commands will set the selected cells to 0. This implies that we can keep issuing write commands until all bits are zeroed out. This significantly reduces the number of erase cycles required and thus increases the memory's longevity.

Our encoding scheme works in two steps and is displayed in Figure 3. First, we apply a Gray code encoding of the monotonic counter used in Section 4.1. In the second step each bit of the resulting Gray code is stored over b blocks of p pages each containing c memory cells and encoded as the number of bits set modulo 2. Each bit flip of the Gray code thus only requires a single write command to one of the pages, which will only touch a single memory cell.

At this low level we must also take unexpected loss of power into consideration; a write/erase command may be interrupted. For write commands this is a non-issue. Since they only affect a single memory cell (t.i. 1 bit), loss of power during their execution will always have a similar effect as a loss of power before or after the command was issued. Both cases are handled at the higher level of the algorithm.

Erase commands in contrast, may not be atomic. Loss of power may leave a memory block in an inconsistent

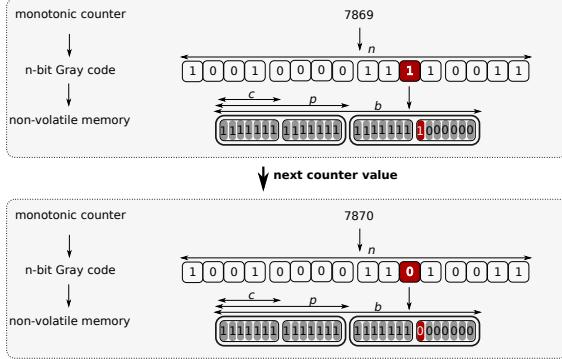


Figure 3: We optimize for write-limited non-volatile memory in two steps: (1) encoding the monotonic counter as an n -bit Gray code and (2) storing each bit in b blocks of p pages each containing c memory cells.

state and gaps in the bit pattern shown in Figure 3 may emerge. As memory content is aggregated to a single bit of a Gray code, such events are at the level of the Ariadne algorithm similar to a crash of the system before or after the counter was incremented. We only need to ensure that each (non-interrupted) write command flips a bit. This can be achieved easily by keeping a local copy of non-volatile memory that is read during the recovery phase.

5 Security Evaluation

Rollback prevention To verify that Ariadne ensures state-continuous execution, we modeled the execution of a module φ :

$$\varphi(\text{State}\varphi, \text{Input}) \rightarrow \text{State}\varphi$$

and proved that even under attack steps, φ is never called with a stale state, or with the same state with different input.

More specifically, we modeled the state of a machine \mathcal{S} as a tuple (H, C, t, P, φ, g) containing a set of packages written to disk H and a monotonic counter C . A term t represents a small program that calls the Ariadne protected module φ in an infinite loop. A set of packages P is kept representing all packages generated by Ariadne. Ghost state g keeps track of (g_{state}, g_i) , the last module state and input that was provided to φ as input.

Based on the state space \mathcal{S} , we built a state machine with a step relation $S_M \subseteq \mathcal{S} \times \mathcal{S}$ where every step is

either a program step or an attack step:

$$S_M = \{(s, s') \in \mathcal{S} \times \mathcal{S} \mid \begin{aligned} &\text{program_step } s \text{ } s' \\ &\vee \text{inc_counter_step } s \text{ } s' \\ &\vee \text{modify_hdd_step } s \text{ } s' \\ &\vee \text{crash } s \text{ } s' \end{aligned}\}$$

Program steps simply take one evaluation step of term t . Attack steps on the other hand include:

- **inc_counter_step s s'** : An attacker may advance the monotonic counter. This may prevent the module from ever resuming its state, but it must not break state continuity.
- **modify_hdd_step s s'** : An attacker can modify the contents of the hard disk drive. When the program t reads a package from disk after a crash, it may not receive the last, fresh package.
- **crash s s'** : The system may crash at any point in time. This will immediately reset the program t to its initial state.

Our proof uses rely-guarantee reasoning [20] to show that starting from a known good state s_0 and taking any number of steps in S_M , we will only take allowed steps. A step is allowed when the module φ is called with as input state $(\varphi(s))(g(s))$, the resulting state of the last call to φ . By definition not calling the module $(g'(s) = g(s))$ or purging the module $(g(s') = (s_0, i_0))$ are also allowed:

$$A = \{(s, s') \in S_M \mid \begin{aligned} &\text{state}(g(s')) = (\varphi(s))(g(s)) \\ &\vee g'(s) = g(s) \\ &\vee g(s') = (s_0, i_0) \end{aligned}\}$$

In total the proof³ consists of 74 definitions, 74 lemmas and totals 4,823 lines. The optimizations proposed in Sections 4.2 and 4.3 were not modeled.

Liveness property Recall that we also required that an unexpected crash should never let the system end up in a state where it could never advance from. This property is trivially met: the system can only get stuck when it requires access to a package it did not yet store. Since Ariadne ensures that the package with the next counter value is always committed to disk before the counter in non-volatile memory is updated, such situations can never occur.

³The proof is available for download at <https://distrinet.cs.kuleuven.be/software/sce/ariadne.html>

6 Applications

We already showed that by relying only on a monotonic counter, we can easily optimize our solution, but which optimizations are best applied, depends heavily on the platform and the security guarantees required. We provide two examples, both on the x86 platform.

6.1 TPM NVRAM to Store Freshness Information

On the x86 platform the TPM chip provides an obvious location to store freshness information: it offers many security primitives, is already widely available on commodity devices and is secure against all but sophisticated hardware attacks. Especially the latter is important in a digital rights management (DRM) setting where the client may not be fully trusted, or when the device may get physically stolen by an attacker.

6.1.1 Platform Considerations

Using the TPM, we have two options. We could use the monotonic counters that are directly provided by the TPM to implement the basic Ariadne algorithm from Section 4.1. Unfortunately, in order to ensure that “[the counter does] not wear out in the first 7 years of operation”, TPM chips may throttle the speed at which it may be incremented. To comply to the TPMv1.2 specification [53], TPMs only “must be able to increment at least once every 5 seconds.” Fortunately TPMs already report on their throttling mechanism, but timeouts between increments may render it unacceptable for some use cases.

Alternatively, we could use TPM NVRAM to store the monotonic counter. While the specification does not require writing operations to be throttled, many TPM implementations rely on EEPROM for NVRAM storage [4, 43, 44], which may cause repeated NVRAM write operations to “prematurely wear out the TPM.” [53] The counter encodings presented in Sections 4.2 and 4.3 can optimize for longevity of this type of memory, but TPM firmware may need to be updated to report the type of memory used, their access granularity and the number of program/erase cycles that are supported.

Independent of whether we use the TPM’s native monotonic counters or NVRAM, the number of supported counters is extremely limited. We take the same approach as related work [32, 46] and introduce an indirection. Only a unique *Theseus* module will access the TPM chip directly to store freshness information. Other modules link to a `libariadne_n` library – a slightly modified version of `libariadne` – that uses the *Theseus* instance to (state-continuously) store a monotonic counter. We elaborate on this construct in Appendix B.

Care must be taken that only a single instance of the *Theseus* module exists at any point in time. Failure to do so may enable race conditions on the monotonic counter and the same module state may be recovered by multiple module instances. From that point on, an attacker is able to break state continuity trivially by providing different input to the various instances with identical state. Many protected-module architectures [8, 30, 48] allow modules to access their module ID; a unique ID per boot cycle starting at value 0. This makes it trivial to ensure that only a single (*Theseus*) module exists; if *Theseus* at initialization-time determines that it received an ID different from zero, it could simply abort.

Intel SGX is a particular case that does not provide such functionality. In this case a static TPM PCR register could be used instead. When the *Theseus* module starts, it could first extend a static PCR (e.g., PCR 8) with a random value. As static PCRs can never be set to a specific value and only be reset by rebooting the platform, any deviation of the expected resulting value would indicate that a previous instance may have started already.

6.1.2 Implementation

We implemented⁴ our prototype on top of Fides [48], an open-source, hypervisor-based protected-module architecture. Table 1 displays the breakdown of the *Theseus* module and the `libariadne_n` library. It is shown that Ariadne’s algorithm is fairly small with only 503 LoCs [56]. The use of balanced Gray codes adds 908 LoCs in total of which 583 LoCs are used to provide static, precomputed metadata. As could be expected, the implementation to balance writes to non-volatile memory is with only 163 LoCs much smaller. Most source code is required to access the TPM chip (1,934 LoCs), perform cryptographic computations (3,237 LoCs), or use basic functions on top of Fides (3,442 LoCs). This results in a total line count for *Theseus* of 10,399 LoCs. `libariadne_n` is with 7,291 LoCs a bit smaller.

6.1.3 Performance Evaluation

To benchmark TPM operations, `libariadne_n` and the *Theseus* module, we used a Dell Optiplex 7010 desktop system. It is equipped with an Intel Core i7-3770 CPU (Ivy Bridge) running at 3.40GHz, a TPMv1.2 chip and an SSD drive. It used the generic 3.16.0-31 Linux kernel.

TPM Microbenchmarks We performed benchmarks of various TPM operations (see Figure 4). To force the TPM into a defensive mode and protect itself against possibly wearing out non-volatile memory, we executed

⁴Our prototype is available for download at <https://distrinet.cs.kuleuven.be/software/sce/ariadne.html>

libariadne_n	C	x86-64
Fides_tools	1,687	1,755
libcrypto	1,661	1,576
libariadne.base	503	0
other	109	0
<i>Total</i>	<i>3,960</i>	<i>3,331</i>

Theseus	C	x86-64
Fides_tools	1,687	1,755
libcrypto	1,661	1,576
libariadne.base	503	0
libtpm	1,934	10
libnv_optimize	163	0
libgray_codes	908	0
other	202	0
<i>Total</i>	<i>7,058</i>	<i>3,341</i>

Table 1: Breakdown of the source code of **libariadne_n** and **Theseus**.

every operation 1,050 times. The first 50 timings were later discarded to avoid recording timing results before any defense mechanism kicked in. As expected, commands that do not require access to non-volatile memory, performed well. Polling the TPM for hardware specific information using a TPM.GetCapability command finished with only 12.00ms (stdev 2.43) per command significantly faster than any other command. Reading and extending a PCR value took with 24.00ms twice as long (stdev 4.55 and 0.01, resp.).

Other commands such as incrementing a monotonic counter and reading and writing to TPM NVRAM require the creation of an OIAP session. For each benchmark we created a single authorization session and kept the session open for the entire benchmark. Creating and closing a session are thus not included in the measurements, but cost 24.05ms (stdev 0.66) and 23.95ms (stdev 0.01) respectively. Incrementing the same monotonic counter 1,050 times without any interruption between increments, cost with 95.99ms (stdev 5.79) significantly more. To determine whether its performance was throttled to protect against wearing out, we re-executed the same benchmark but with a 5 seconds interval between increments. As expected given that the TPM reports that it does not throttle its speed, the new benchmark provided similar results (95.91ms/inc, stdev 5.41).

We performed similar benchmarks for writing to TPM NVRAM. Each write command only wrote a single zero byte and finished in 144.00ms (stdev 4.26). Introducing a 5 second interval between two write operations did not increase performance and finished in 143.91ms (stdev 3.96). Even though none of these recorded write operations actually required physical writes to TPM NVRAM

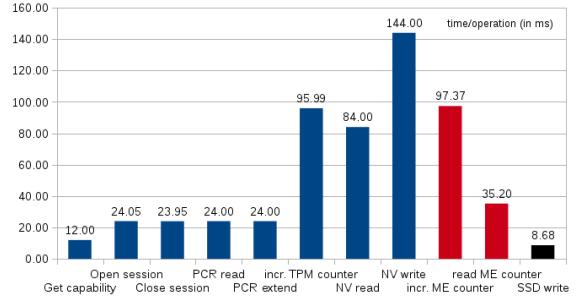


Figure 4: Microbenchmark results (in ms) of various TPM, ME and SSD operations.

as the same data was provided in every command, reading from TPM NVRAM performed much better at “only” 84.00ms (stdev 2.11). The extremely slow operation of our TPM chips is best illustrated by the performance of the SSD drive. Creating a 1 byte file is with only 8.68 ms (stdev 3.08) significantly faster than accessing TPM NVRAM.

Incrementing a monotonic counter is much faster than writing to TPM NVRAM. We attribute this difference to a wear leveling mechanism within the TPM chip. We tested the presence of such a mechanism on an unused Broadcom TPM chip. We allocated 4 regions of 64 bytes and wrote intermittent 0x00 and 0xff bytes until these regions failed to retain their contents. Regions broke respectively after 1,450K, 621K, 493K and 301K writes. Without the presence of a wear leveling mechanism, we would have expected that all regions would have failed after an equal amount of write cycles.

In Sections 4.2 and 4.3 we showed that we can minimize the number of program/erase cycles required to implement a counter. Most importantly, we showed that a 2-phase commit protocol to address sudden loss of power during increments is not required. Using these optimizations, we expect that we can further reduce the time required to increment a monotonic counter. We recommend TPM vendors to enable TPM owners to take responsibility of TPM NVRAM wear leveling and expose write/erase commands explicitly.

Benchmarking Theseus and libariadne_n We implemented a state-continuous module that keeps track of a virtual counter. The module was advanced 1,050 times, and we again discarded the first 50 timing results.

When this module accessed TPM NVRAM directly, advancing the counter took 152.01ms (stdev 3.73) in total (see Table 2). As expected, updating freshness information in TPM NVRAM is with 93.68% of the total time by far the most time-costly operation. Calculating the next Gray code, creating the package to store on disk

(in ms)	TPM	SSD	Comp.
Virt. Cntr. (NVRAM)	142.40	9.36	0.25
Virt. Cntr. (Thesius)	145.68	21.91	0.40

Table 2: Benchmark results of Theseus and libariadne.n

and the overhead introduced by our protected-module architecture, is with 0.25ms negligible.

When the same module used the Theseus module to store freshness information, performance was impacted marginally. Writing the state update of Theseus to TPM NVRAM is with 145.68ms still responsible for 86.72% of the total time required. Communication with the Theseus module caused the time attributed to computation to increase to 0.40ms, but remains negligible. Now that not one but two packages need to be stored on the SSD drive, time lost due to SSD overhead increased from 9.36ms to 21.91ms.

6.2 Intel ME to Provide State-Continuity to Intel SGX

In contrast to earlier provided specifications [18], it became clear with the publication of the Intel SGX SDK [19] in December 2015, that SGX enclaves *can* easily access monotonic counters. It appears that these counters are stored on the management engine (ME) [37], not on non-volatile memory within the CPU package.

Platform Considerations With Intel ME readily available on recent Intel systems it is an interesting location to keep freshness information. Unfortunately it comes with significant downsides compared to an SGX/TPM approach, at least from an academic point of view. First, Intel ME uses a separate processor on the platform control hub (PCH) running its own kernel and processes (e.g., Intel AMT, EPID, etc.). SGX enclaves can access the ME by calling a platform-specific enclave (PSE) that uploads a Java applet through the generic Dynamic Application Loader (DAL) interface [37]. Unfortunately, this re-introduces a TCB of probably a considerable size.

Second, it is unclear how the ME is protected against physical attacks, one of the key selling points of Intel SGX. Related work showed [57] that the ME firmware is only integrity protected and may be accessed through physical attacks [52]. It is unclear how sensitive data stored in the ME is replay protected and whether additional security measures were added in the last generation of PCHs.

Unfortunately Intel SGX/ME does not provide a mechanism for enclaves to determine whether they are the first/only instance. To protect against forking-attacks,

we must make sure that only a single instance of a state-continuous enclave is running. Similarly to the Fides/TPM platform, we could use a PCR register to detect the presence of other enclaves. On systems that lack a discrete, hardware TPM chip, similar functionality is provided by the ME engine.

This approach has the obvious disadvantage that two enclaves need to store their state on disk for every state update. Benchmarks of the Fides/TPM architecture (Section 6.1.3) show that this results in a non-negligible performance impact. This additional overhead could easily be avoided when a (volatile) in-use bit would be kept with the ME monotonic counters. When the enclave state is being recovered, Ariadne should check this bit: when the counter is not in use, the enclave should set this bit – using an atomic `test_and_set` operation – and continue its recovery. When the bit is already set, another instance of the same enclave may still be running and the newly created instance of the enclave should be destroyed to prevent forking attacks. The Theseus module described in Section B uses the same approach.

Benchmarks In the previous section we determined that on the Fides/TPM architecture at least 99.76% of Ariadne’s execution time was spent on TPM and disk accesses. We are thus especially interested in the cost of ME monotonic counter increments. We performed microbenchmarks on a recent Dell Inspiron 13 7359 equipped with a Skylake Core i7-6500U processor running Windows 10. As with TPM benchmarks, we discarded the first 50 calls to account for warm-up time for each test. Calling an enclave that returned a static integer value 1,050 times took 0.013ms (stdev 0.003). When the enclave established a single PSE session and incremented a monotonic counter upon each call, performance decreased to 97.38ms (stdev 21.04) with outliers ranging between 71.34 and 251.66ms. Time required to increment an ME monotonic counter is thus comparable to incrementing a monotonic counter in our TPM chip (95.99ms, see Figure 4). When the ME counter was only read, not incremented, performance increased significantly to 35.21ms (stdev 1.17). This leads us to expect that the slow operation of counter increments may be attributed to a throttling mechanism to avoid wearing out ME non-volatile memory.

7 Related Work

Many security architectures have been proposed in recent years. Some rely on a huge TCB making state continuity a much easier problem to solve. Other architectures have ignored the problem altogether and are susceptible to rollback attacks and/or cannot guarantee that the sys-

tem will always be able to advance after power is lost. State-continuity is a problem that has not gained a lot of research attention. Only Parno et al. [32] and we in earlier work [46] proposed solutions to the problem. Others provided more application-specific approaches.

7.1 Systems with a Large TCB

Many systems require state-continuity guarantees in one way or another. These include early designs of protected-module architectures such as Terra [15], AppCores [41] and Proxos [51], but also more conventional systems. Cloud providers must also ensure that the entire state of a virtual machine [16, 58] cannot be rolled back. Similarly, applications on modern operating systems must be restarted from their most recent state.

State-continuity on such platforms can be easily provided as the kernel and/or hypervisor is trusted to isolate disk accesses. It is assumed that an attacker cannot gain access to previously stored states and thus also cannot roll back the state. Since this assumption relies on the correct implementation of the access control logic, this is hard to guarantee in practice. Interestingly, the same applies to many formally verified systems such as seL4 [21], HyperV [24] and XMHF [54]. While the most privileged layer is verified, the file system’s implementation usually is not and these systems are still susceptible to attack. Obviously, such designs can also not defend against hardware-level attacks where an attacker is able to physically access the disk to copy and restore stale states.

7.2 Hardware Modifications

Systems such as XOM [26] try to defend against attackers snooping on memory buses by confidentiality and integrity protecting data before it is stored in main memory. Suh et al. [50] show that without anti-replay protection, stale memory contents could be presented as being fresh. Their Aegis architecture and subsequent platforms [9, 29] defend against such attacks by including a freshness tag.

Defense mechanisms against memory replay attacks, do not have to take unexpected loss of power into consideration, nor do they have to consider limitations of non-volatile memory. This enables very different approaches.

7.3 Protected-Module Architectures

Many protected-module architectures have been proposed over the recent years. [9, 22, 27, 28, 30, 31, 38, 41, 48] Interestingly, many do not address the state-continuity problem. To the best of our knowledge only two papers directly addressed the problem. Parno et al. [32] were the first to highlight the problem. They proposed

two mechanisms to provide state-continuity guarantees. Memoir-Basic uses a cryptographic hash to keep track of the fresh state. Unfortunately storing this freshness tag in TPM NVRAM would wear out its memory too quickly to be of any practical use. The authors acknowledge this limitation and propose a solution. Memoir-Opt uses a similar freshness tag, but it relies on TPM PCR registers to protect the most recent value while the system is up and running. When power is lost unexpectedly, an uninterruptible power source ensures that it is written to TPM NVRAM.

In earlier work we proposed ICE [46, 47], an alternative design that assumes “guarded memory”: volatile memory within the CPU package that is written to untrusted, non-volatile memory when power is lost. By avoiding TPM/ME accesses for each state update altogether, we achieve better performance results.

Both Memoir and ICE avoid the significantly constrained TPM NVRAM and rely on some kind of uninterruptible power source to provide state-continuity guarantees. While this leads to better performance, these solutions cannot be readily applied in practice. In contrast Ariadne can be used to protect stateful enclaves against rollback attacks on commodity devices.

7.4 Special-Purpose Applications

Many applications have been proposed that provide special-purpose solutions. We can easily provide similar guarantees, but in a much more general way.

Chun et al. proposed a construct called append-only memory [10] to prevent nodes in a distributed network from making conflicting claims to different nodes. A practical implementation was left as future work. In subsequent work Levin et al. [25] provided similar guarantees by only relying on a trusted incrementer (TrInc) that is able to locally store attestation request of monotonic counters.

Schellekens et al. [39] addressed the problem of limited TPM NVRAM. They show that sensitive data can be stored in non-volatile memory off-chip. A light-weight authentication protocol ensures a secure channel between the trusted module and untrusted, non-volatile memory. However, a monotonic counter needs to be incremented for each write instruction and stored in the modified TPM chip. How this counter can be stored efficiently, is not discussed. We achieve similar results without requiring any modification to the TPM chip.

Kotla et al. [23] propose a system to enable offline use of sensitive data. Once sensitive data is accessed, it cannot be denied by the user. Alternatively, if the user attests that the data was never accessed, she will no longer be able to do so in the future. Interestingly, only a very limited TCB needs to be trusted.

The Intel SGX SDK Manual for Windows [19] also discusses how enclaves can be used to implement limited-use policies. Their approach is similar to the inc-then-store discussed in Section A.1. While their approach guarantees that enclave states cannot be rolled back, they fail to guarantee continuous execution of an enclave and liveness of the system. We provide stronger guarantees in a more general way.

8 Conclusion

Protected module architectures enable protected module writers to guarantee formally provable security properties of their code while the system is up and running. But without support for state-continuity, stateful modules are prone to rollback attacks.

Existing solutions relied on the irreversibility property of hash functions and required an uninterruptible power source or risked wearing out non-volatile TPM NVRAM.

We presented Ariadne, the first solution that achieves state continuity based on a counter. By relying on balanced Gray codes to encode this counter, we achieved the theoretical lower limit of requiring only a single bit flip per state update.

Embedded devices can use Ariadne to minimize their total bill of materials. On SGX-enabled x86 platforms Ariadne can be readily applied by relying on the TPM’s monotonic counters, TPM NVRAM or the management engine’s monotonic counters. We showed that the choice depends on the specific TPM chip on the platform, attack model and use case.

Acknowledgments

This work has been supported in part by the Intel Lab’s University Research Office, by the Research Fund KU Leuven, and by the Research Foundation - Flanders (FWO).

References

- [1] AGTEN, P., JACOBS, B., AND PIJSESENS, F. Sound modular verification of C code executing in an unverified context. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’15)* (Jan. 2015).
- [2] AGTEN, P., STRACKX, R., JACOBS, B., AND PIJSESENS, F. Secure compilation to modern processors. In *Proceedings of the 25th Computer Security Foundations Symposium* (Los Alamitos, CA, USA, 2012), CSF’12, IEEE Computer Society, pp. 171–185.
- [3] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), vol. 13 of *HASP’13*, ACM.
- [4] ATMEL. At97sc3204. <http://www.atmel.com/images/atmel-5295s-tpm-at97sc3204-lpc-interface-datasheet-summary.pdf>.
- [5] AVONDS, N., STRACKX, R., AGTEN, P., AND PIJSESENS, F. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *Security and Privacy in Communication Networks (SecureComm’13)* (Sept. 2013), T. Zia, A. Zomaya, V. Varadarajan, and M. Mao, Eds., vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer International Publishing, pp. 252–269.
- [6] AZAB, A., NING, P., AND ZHANG, X. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS’11, ACM.
- [7] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI’14)* (2014).
- [8] BRASSER, F., EL MAHJOUB, B., SADEGHI, A.-R., WACHSMANN, C., AND KOEBERL, P. Tytan: Tiny trust anchor for tiny devices. In *Proceedings of the 52Nd Annual Design Automation Conference* (New York, NY, USA, 2015), DAC’15, ACM.
- [9] CHAMPAGNE, D., AND LEE, R. Scalable architectural support for trusted software. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture* (2010), HPCA’10, IEEE Computer Society, pp. 1–12.
- [10] CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP’07, ACM, pp. 189–204.
- [11] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. Cryptology ePrint Archive, Report 2015/564, 2015.
- [12] EL DEFRAWY, K., AURÉLIEN FRANCILLON, D., AND TSUDIK, G. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Proceedings of the Network & Distributed System Security Symposium* (Feb. 2012), NDSS’12.
- [13] EVTYUSHKIN, D., ELWELL, J., OZSOY, M., PONOMAREV, D., GHAZALEH, N. A., AND RILEY, R. Iso-X: A flexible architecture for hardware-managed isolated execution. In *47th Annual IEEE/ACM International Symposium on Microarchitecture* (Dec. 2014).
- [14] FLAHIVE, M. Balancing cyclic R-ary Gray codes II. *The Electronic Journal of Combinatorics* 15 (2008), R128.
- [15] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Operating Systems Review* (New York, NY, USA, 2003), vol. 37 of *OSR’03*, ACM, pp. 193–206.
- [16] GARFINKEL, T., AND ROSENBLUM, M. When virtual is harder than real: security challenges in virtual machine based computing environments. In *Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), HOTOS’05, USENIX Association, pp. 20–25.
- [17] HOOGSTRATEN, H., PRINS, R., NIGGEBRUGGE, D., HEPENER, D., GROENEWEGEN, F., WETTINCK, J., STROOY, K., ARENDTS, P., POLS, P., KOUPRIE, R., MOORREES, S., VANPELT, X., AND HU, Y. Z. Black Tulip - report of the investigation into the DigiNotar certificate authority breach. Tech. rep., FoxIT, 2012.
- [18] INTEL CORPORATION. *Software Guard Extensions Programming Reference*, 2013.

- [19] INTEL CORPORATION. *Intel Software Guard Extensions Evaluation SDK for Windows OS*, 2015.
- [20] JONES, C. Tentative steps toward a development method for interfering programs. In *ACM Transactions on Programming Languages and Systems (TOPLAS)* (New York, NY, USA, 1983), vol. 5, ACM, pp. 596–619.
- [21] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP’09, ACM.
- [22] KOEBERL, P., SCHULZ, S., SADEGHI, A.-R., AND VARADHARAJAN, V. Trustlite: a security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys’14, ACM, p. 10.
- [23] KOTLA, R., RODEHEFFER, T., ROY, I., STUEDI, P., AND WESTER, B. Pasture: secure offline data access using commodity trusted hardware. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), OSDI’12, USENIX Association, pp. 321–334.
- [24] LEINENBACH, D., AND SANTEL, T. Verifying the microsoft hyper-v hypervisor with vcc. In *FM 2009: Formal Methods*. Springer, 2009, pp. 806–809.
- [25] LEVIN, D., DOUCEUR, J. R., LORCH, J. R., AND MOSCIBRODA, T. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2009), vol. 9 of NSDI’09, USENIX Association, pp. 1–14.
- [26] LIE, D., CHANDRAMOHAN, T., MARK, M., PATRICK, L., DAN, B., JOHN, M., AND MARK, H. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (2000), vol. 35 of ASPLOS’00, ACM, pp. 168–177.
- [27] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (Washington, DC, USA, May 2010), S&P’10, IEEE Computer Society, pp. 143–158.
- [28] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems* (Apr. 2008), EuroSys’08, ACM.
- [29] McKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP’13, ACM, p. 8.
- [30] NOORMAN, J., AGTEN, P., DANIELS, W., STRACKX, R., HERREWEGE, A. V., HUYGENS, C., PRENEEL, B., VERBAUWHEDE, I., AND PIJSESENS, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium* (2013).
- [31] OWUSU, E., GUAJARDO, J., MCCUNE, J., NEWSOME, J., PERRIG, A., AND VASUDEVAN, A. OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), CCS’13, ACM, pp. 13–24.
- [32] PARNO, B., LORCH, J. R., DOUCEUR, J. R., MICKENS, J., AND MCCUNE, J. M. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2011), S&P’11, IEEE, pp. 379–394.
- [33] PATRIGNANI, M., AGTEN, P., STRACKX, R., JACOBS, B., CLARKE, D., AND PIJSESENS, F. Secure compilation to protected module architectures. In *Transactions on Programming Languages and Systems (TOPLAS)* (New York, NY, USA, Apr. 2015), vol. 37, ACM, pp. 6:1–6:50.
- [34] PATRIGNANI, M., CLARKE, D., AND PIJSESENS, F. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS’13)* (2013), C.-c. Shan, Ed., vol. 8301 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 176–191.
- [35] RAJ, H., SAROIU, S., WOLMAN, A., AIGNER, R., COX, J., ENGLAND, P., FENNER, C., KINSHUMANN, K., LOESER, J., MATTOON, D., NYSTROM, M., ROBINSON, D., SPIGER, R., THOM, S., AND WOOTEN, D. fTPM: A firmware-based TPM 2.0 implementation. Tech. Rep. MSR-TR-2015-84, Microsoft, Nov. 2015.
- [36] RAJ, H., SAROIU, S., WOLMAN, A., AIGNER, R., COX, J., ENGLAND, P., FENNER, C., KINSHUMANN, K., LOESER, J., MATTOON, D., NYSTROM, M., ROBINSON, D., SPIGER, R., THOM, S., AND WOOTEN, D. fTPM: A firmware-based TPM 2.0 implementation. In *Proceedings of the 25th USENIX security symposium* (Aug. 2016), SSYM’16, USENIX Association.
- [37] RUAN, X. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*, 1 ed., vol. 1. Apress, 2014.
- [38] SAHITA, R., WARRIOR, U., AND DEWAN, P. Protecting Critical Applications on Mobile Platforms. *Intel Technology Journal* 13, 2 (June 2009), 16–35.
- [39] SCHELLEKENS, D., TUYLS, P., AND PRENEEL, B. Embedded trusted computing with authenticated non-volatile memory. In *First International Conference on Trusted Computing and Trust in Information Technologies (TRUST’08)* (2008), P. Lipp, A.-R. Sadeghi, and K.-M. Koch, Eds., Lecture Notes in Computer Science, Springer-Verlag, pp. 60–74.
- [40] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy data analytics in the cloud using SGX. In *36th IEEE Symposium on Security and Privacy* (May 2015), IEEE Institute of Electrical and Electronics Engineers.
- [41] SINGARAVELU, L., PU, C., HÄRTIG, H., AND HELMUTH, C. Reducing TCB complexity for security-sensitive applications: three case studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (New York, NY, USA, 2006), EuroSys’06, ACM, pp. 161–174.
- [42] SOLID STATE STORAGE INITIATIVE. NAND flash solid state storage for the enterprise – an in-depth look at reliability. http://www.vikingtechnology.com/uploads/NV_DIMM_ROI.pdf.
- [43] STMICROELECTRONICS. St19np18-tpm. <http://www.bdtic.com/DownLoad/ST/ST19NP18-TPM.pdf>.
- [44] STMICROELECTRONICS. St33tpm12lpc. <http://datasheet.octopart.com/ST33ZP24AR28PVSP-STMicroelectronics-datasheet-16348175.pdf>.
- [45] STRACKX, R., AGTEN, P., AVONDS, N., AND PIJSESENS, F. Salus: Kernel support for secure process compartments. In *Endorsed Transactions on Security and Safety* (2015), vol. 15, ICST.
- [46] STRACKX, R., JACOBS, B., AND PIJSESENS, F. ICE: A passive, high-speed, state-continuity scheme. In *Annual Computer Security Applications Conference* (2014), ACSAC’14.
- [47] STRACKX, R., JACOBS, B., AND PIJSESENS, F. ICE: A passive, high-speed, state-continuity scheme (extended version). CW Reports CW672, Department of Computer Science, KU Leuven, September 2014.

- [48] STRACKX, R., AND PIJSESENS, F. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM conference on Computer and Communications Security* (New York, NY, USA, October 2012), CCS’12, ACM, pp. 2–13.
- [49] STRACKX, R., PIJSESENS, F., AND PRENEEL, B. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *Security and Privacy in Communication Networks (SecureComm’10)* (2010), S. Jajodia and J. Zhou, Eds., vol. 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer Berlin Heidelberg.
- [50] SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing* (New York, NY, USA, 2003), ICS’03, ACM, pp. 160–171.
- [51] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI’06, USENIX Association, pp. 279–292.
- [52] TERESHKIN, A., AND WOJTCZUK, R. Introducing ring -3 rootkits. In *Black Hat USA* (July 2009).
- [53] TRUSTED COMPUTING GROUP. *Design Principles Specification Version 1.2*, 2011.
- [54] VASUDEVAN, A., CHAKI, S., JIA, L., MCCUNE, J., NEWSOME, J., AND DATTA, A. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), S&P’13, IEEE Computer Society, pp. 430–444.
- [55] WANG, Y., KEI YU, W., XU, S., KAN, E., AND SUH, G. Hiding information in flash memory. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 271–285.
- [56] WHEELER, D. A. SLOCCount. <http://www.dwheeler.com/sloccount/>.
- [57] WOJTCZUK, R., AND TERESHKIN, A. Attacking intel BIOS. In *Black Hat USA* (July 2009).
- [58] XIA, Y., LIU, Y., CHEN, H., AND ZANG, B. Defending against vm rollback attack. In *42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)* (June 2012).
- [59] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *36th Symposium on Security and Privacy* (May 2015).

A State-Continuity Based on a Monotonic Counter: Strawman Attempts

State continuity is a hard problem. Solutions borrowed directly from anti-replay defenses fail to provide the required security guarantees. We discuss three approaches.

A.1 Inc, then Store

Listing 3 displays a first – but flawed – implementation. Its implementation is straightforward. Recall the PIN-protected module from Section 3.3. When a user tries to retrieve the module’s secret, she calls the `get_secret` entry point and supplies a PIN. Before this

```

1 #include <libariadne/interface.h>
2
3 void store_state( Blob *blob, String f_format ){
4     hwnctr.inc();
5     hdd.write( create_pkg(blob, hwnctr.value()), f_format );
6 }
7
8 Blob *retrieve_state( String f_format ){
9     Package *pkg = hdd.read( f_format, hwnctr.value() );
10
11    if ( pkg == NULL || !auth( pkg, get_mac_key() ) )
12        return NULL;
13
14    if ( pkg->cntr != hwnctr.value() )
15        return NULL;
16
17    return decrypt( pkg, get_enc_key() );
18 }
19
20 void purge_state( Blob * init_blob, String f_format ){...}

```

Listing 3: Incrementing the counter before writing the package to disk will fail to guarantee liveness.

PIN is checked, the module collects the state of the module and passes it together with the provided PIN and called function, to the `store_state` library function. There the (hardware) monotonic counter is incremented first (listing 3, line 4). Afterwards a new package is created containing the incremented counter value and written to disk.

When the system is rebooted and the module needs to be reloaded in memory, the `on_load` function is called implicitly (listing 1, line 7) which in turn calls the `retrieve_state` library function. Next, the library searches for the filename of the last package written to disk. When this package is read and its integrity verified, the counter value enclosed in the package is compared to the hardware monotonic counter (listing 3, line 9 - 14). If both counter values match, the package is determined fresh. After decryption, it is returned to the `on_load` module function where the module’s state is restored and the execution of the called function is restarted (listing 1, line 11). If the read package from disk is not fresh or its integrity check failed, `NULL` is returned and the module is `reset` losing the stored secret indefinitely.

Attack 1: Breaking Liveness

Unfortunately, the provided scheme is flawed. Imagine an unexpected loss of power immediately after the monotonic counter was incremented (i.e., after listing 3, line 4). When the module is reloaded, the `retrieve_state` library function will only accept packages containing a counter value equal to the hardware counter as being fresh. But this package was never stored and the sys-

```

1 #include <libariadne/interface.h>
2
3 void store_state( Blob *blob, String f_format ){
4     hdd.write( create_pkg( blob, hwcntr.value() + 1 ),
5                 f_format );
6     hwcntr.inc();
7 }
8
8 Blob *retrieve_state( String f_format ){
9     Package *pkg = hdd.read( f_format, hwcntr.value() );
10
11    if ( pkg == NULL || !auth( pkg, get_mac_key() ) )
12        return NULL;
13
14    if ( pkg->cnt != hwcntr.value() )
15        return NULL;
16
17    return decrypt( pkg, get_enc_key() );
18 }
19
20 void purge_state( Blob * init_blob, String f_format ){...}

```

Listing 4: Writing a package to disk before incrementing the monotonic counter will leave the system susceptible to dictionary-style attacks.

tem ends up in a state from where it can never advance. As this situation may also occur even when the system is not under attack, this breaks our liveness requirement. ■

A.2 Store, then Inc

One may be tempted to quickly fix the design flaw of the scheme presented in the previous section by ensuring that packages are written to disk *before* the hardware monotonic counter is incremented. We will show that also this design does not guarantee the required security properties.

Listing 4 displays the updated scheme. When the module requests the libariadne module to store a new state, the `store_state` function will first create a new package with the next counter value and write it to disk (line 4-5) before she increments the hardware counter. The `retrieve_state` function is unchanged.

With these changes in place, a package has always been written to disk that matches the value of the monotonic counter. Therefore the module is always able to recover its state under benign conditions and liveness of this scheme is ensured. Unfortunately, this mitigation enables an attacker to execute a dictionary-style attack.

Attack 2: Dictionary Attack

Let's reuse the module from Section 3.3 as an example. Assume that the attack starts when the module still grants

the attacker with three attempts to guess the correct PIN, and that the monotonic counter has value c . Let's call this state s_0 .

In the preliminary step of the attack, the attacker iterates over all entries of her dictionary. For each PIN, she calls the `get_secret` entry point of the module, but crashes the system after the new package – enclosed counter value $c + 1$ – has been written to disk (listing 4, line 4). Since the module could not yet commit to the new input by incrementing the monotonic counter, the module will recover its previous state and once again wait for user input from state s_0 . At that point, the attacker continues the same process with the next entry in her dictionary.

When the attacker has reached the end of her dictionary, she finally allows the module to commit to the provided input by incrementing the monotonic value to $c + 1$. Afterwards she crashes the system again.

This marks the start of the second step of her attack. When the module is reloaded in memory, it needs to retrieve its state from disk. Its integrity and freshness is verified before it is used to resume its state. But for every dictionary entry, the attacker possesses a package that will be accepted as being authentic and fresh. She completes her dictionary attack by selecting any package and let the module recover its state based on the enclosed guessed PIN. When she learns the PIN is incorrect, she crashes the system, and selects another “fresh” package. ■

A.3 Inc once during recovery

In order to defend against dictionary attacks, one must ensure that no other packages with the same fresh counter value exist when the module is being resumed. Incrementing the counter once during recovery does *not* provide this guarantee. We show an attack against such an implementation.

For completeness we show in Listing 5 the (vulnerable) implementation of the state-continuity security measure. The implementation is identical to that of listing 4, but lines 23 to 25 were added. The main issue is that an attacker is able to abuse the creation and storage of packages during recovery (lines 23 and 24) to keep stale packages fresh. This enables attacks similar to the dictionary attack of Section A.2. Since the attack is much harder to grasp completely, we show a simpler version: an attacker is able to provide a guess of the PIN without it being (permanently) recorded, breaking rollback prevention.

Attack 3: Oblivious Steps

For simplicity say that the PIN-protected module is in a state s_0 with 3 PIN-guesses left and the monotonic

counter is at value 42. The attack is executed in 4 steps. In step I an authentic package is created. This means that an attacker provides a guess ‘‘guess’’, but crashes the module immediately after it wrote the package $\text{pkg}(43 \parallel \text{get_secret} \parallel \text{‘‘guess’'} \parallel s_0)$ to disk (listing 5, line 5). As the module only increments the monotonic counter after the state is stored successfully, it still remains at value 42.

Next in step II, the module is recovered to its previous state, leading to a new package $\text{pkg}(43 \parallel \dots)$ ⁵ and an incremented counter (listing 5 lines 23 to 25). At this moment in time two different packages with enclosed counter value 43 exist.

In step III the implementation of the `retrieve_state` is abused to create a (soon-to-be-fresh) package $\text{pkg}(44 \parallel \dots)$ as follows: First, the module is crashed. During recovery the package $\text{pkg}(43 \parallel \dots)$ is provided and accepted as being fresh. As the `retrieve_state` first writes a new package with an incremented counter value (line 24) the required package is created. Immediately after the package is written to disk, the module is crashed before the monotonic counter can be incremented.

In step IV the module is resumed based on the fresh package $\text{pkg}(43 \parallel \text{get_secret} \parallel \text{‘‘guess’'} \parallel s_0)$ and the monotonic counter is incremented to 44. At this moment the attacker learns the outcome of her guess. If she guessed wrongly, she can choose to crash the system and let the module recover its state from “fresh” package $\text{pkg}(44 \parallel \dots)$. As the module didn’t record her guess, state-continuity is broken. ■

B State-continuous storage for n modules

libariadne as described in Section 4 provides state-continuous storage, at the cost of secure, non-volatile memory. Storing freshness information for every possible protected module in limited-sized, secure non-volatile memory (e.g., TPM NVRAM), is practically infeasible. We resolve the situation using an indirection.

A protected-module “Theseus” is introduced to the system and uses – as the only module in the system – the secure non-volatile memory to store its state. It provides virtual, monotonic counters to other modules executing on the system. Its interface is shown in listing 6.

The `new_counter` entry point to the module, creates a new monotonic counter and protects it with the provided key. It returns the index of the monotonic counter. To

⁵We use the notation $\text{pkg}(43 \parallel \dots)$ here for clarity. We should have introduced a previous state s_{-1} , entrypoint f and input i such that $f(s_{-1}, i) = s_0$. Providing such input to the module will have created the package $\text{pkg}(43 \parallel f \parallel i \parallel s_{-1})$. As f, i and s_{-1} are irrelevant for the attack, we omit these arguments.

```

1 #include <libariadne/interface.h>
2
3 void store_state( Blob *blob, String f_format ){
4     Package *pkg = create_pkg( blob, hwcntr.value() + 1 )
5     hdd.write( pkg, f_format, hwcntr.value() + 1 );
6     hwcntr.inc();
7 }
8
9 Blob *retrieve_state( String f_format ){
10    Package *pkg;
11    Blob *blob;
12
13    pkg = hdd.read( f_format, hwcntr.value() );
14
15    if ( pkg == NULL || !auth( pkg, get_mac_key() ) )
16        return NULL;
17
18    if ( pkg->cntr != hwcntr.value() )
19        return NULL;
20
21    blob = decrypt( pkg, get_enc_key() );
22
23    pkg = create_pkg( blob, hwcntr.value() + 1 );
24    hdd.write( pkg, f_format, hwcntr.value() + 1 );
25    hwcntr.inc();
26
27    return blob;
28 }
29
30 void purge_state( Blob *init_blob, String f_format ){...}

```

Listing 5: When the counter is only incremented once during recovery, an attacker can force the module to keep stale states fresh, enabling rollback attacks.

```

1 int new_counter( uint64_t key );
2 int counter_set_in_use( int idx, uint64_t key, bool in_use );
3 int counter_increment( int idx, uint64_t key );
4 uint64_t counter_value( int idx, uint64_t key );

```

Listing 6: Theseus’ public interface.

protect against inappropriate use, this (index, key)-pair will need to be provided for any subsequent operation on the counter. We assume that communication between the caller and the Theseus module, is confidentiality, integrity and anti-replay protected.

An important feature is the ability to mark a counter to be “in use.” This volatile flag, is used to ensure that only a single instance of a protected module can be resumed after a crash. Hence, `counter_increment` and `counter_value` will return an error code if they are called on a counter that was not previously marked as “in use”.

The Million-Key Question – Investigating the Origins of RSA Public Keys

Petr Švenda, Matúš Nemeč, Peter Sekan, Rudolf Kvašňovský,

David Formánek, David Komárek and Vashek Matyáš

Masaryk University, Czech Republic

Abstract

Can bits of an RSA public key leak information about design and implementation choices such as the prime generation algorithm? We analysed over 60 million freshly generated key pairs from 22 open- and closed-source libraries and from 16 different smartcards, revealing significant leakage. The bias introduced by different choices is sufficiently large to classify a probable library or smartcard with high accuracy based only on the values of public keys. Such a classification can be used to decrease the anonymity set of users of anonymous mailers or operators of linked Tor hidden services, to quickly detect keys from the same vulnerable library or to verify a claim of use of secure hardware by a remote party. The classification of the key origins of more than 10 million RSA-based IPv4 TLS keys and 1.4 million PGP keys also provides an independent estimation of the libraries that are most commonly used to generate the keys found on the Internet.

Our broad inspection provides a sanity check and deep insight regarding which of the recommendations for RSA key pair generation are followed in practice, including closed-source libraries and smartcards¹.

1 Introduction

The RSA key pair generation process is a crucial part of RSA algorithm usage, and there are many existing (and sometimes conflicting) recommendations regarding how to select suitable primes p and q [11, 13, 14, 17, 18] to be later used to compute the private key and public modulus. Once these primes have been selected, modulus computation is very simple: $n = p \cdot q$, with the public exponent usually fixed to the value 65 537. But can the modulus n itself leak information about the design and implementation choices previously used to generate the

primes p and q ? Trivially, the length of the used primes is directly observable. Interestingly, more subtle leakage was also discovered by Mironov [20] for primes generated by the OpenSSL library, which unwantedly avoids small factors of up to 17 863 from $p - 1$ because of a coding omission. Such a property itself is not a security vulnerability (the key space is decreased only negligibly), but it results in sufficiently significant fingerprinting of all generated primes that OpenSSL can be identified as their origin with high confidence. Mironov used this observation to identify the sources of the primes of factorizable keys found by [12]. But can the origins of keys be identified only from the modulus n , even when n cannot be factorized and the values of the corresponding primes are not known?

To answer this question, we generated a large number of RSA key pairs from 22 software libraries (both open-source and closed-source) and 16 different cryptographic smartcards from 6 different manufacturers, exported both the private and public components, and analysed the obtained values in detail. As a result, we identified seven design and implementation decisions that directly fingerprint not only the primes but also the resulting public modulus: 1) Direct manipulation of the primes' highest bits. 2) Use of a specific method to construct strong or provable primes instead of randomly selected or uniformly generated primes. 3) Avoidance of small factors in $p - 1$ and $q - 1$. 4) Requirement for moduli to be Blum integers. 5) Restriction of the primes' bit length. 6) Type of action after candidate prime rejection. 7) Use of another non-traditional algorithm – functionally unknown, but statistically observable.

As different design and implementation choices are made for different libraries and smartcards (cards) with regard to these criteria, a cumulative fingerprint is sufficient to identify a probable key origin even when only the public key modulus is available. The average classification accuracy on the test set was greater than 73% even for a single classified key modulus when a hit within

¹Full details, paper supplementary material, datasets and author contact information can be found at <http://crcs.cz/papers/usenix2016>.

the top 3 matches was accepted². When more keys from the same (unknown) source were classified together, the analysis of as few as ten keys allowed the correct origin to be identified as the top single match in more than 85% of cases. When five keys from the same source were available and a hit within the top 3 matches was accepted, the classification accuracy was over 97%.

We used the proposed probabilistic classifier to classify RSA keys collected from the IPv4 HTTPS/TLS [9], Certificate Transparency [10] and PGP [30] datasets and achieved remarkably close match to the current market share of web servers for TLS dataset.

The optimal and most secure way of generating RSA key pairs is still under discussion. Our wide-scale analysis also provides a sanity check concerning how closely the various recommendations are followed in practice for software libraries and smartcards and what the impact on the resulting prime values is, even when this impact is not observably manifested in the public key value. We identified multiple cases of unnecessarily decreased entropy in the generated keys (although this was not exploitable for practical factorization) and a generic implementation error pattern leading to predictable keys in a small percentage (0.05%) of cases for one type of card.

Surprisingly little has been published regarding how key pairs are generated on cryptographic cards. In the case of open-source libraries such as OpenSSL, one can inspect the source code. However, this option is not available for cards, for which the documentation of the generation algorithm is confidential and neither the source code nor the binary is available for review. To inspect these black-box implementations, we utilized the side channels of time and power consumption (in addition to the exported raw key values). When this side-channel information was combined with the available knowledge and observed characteristics of open-source libraries, the approximate key pair generation process could also be established for these black-box implementations.

This paper is organized as follows: After a brief summary of the RSA cryptosystem, Section 2 describes the methodology used in this study and the dataset of RSA keys collected from software libraries and cryptographic cards. Section 3 provides a discussion of the observed properties of the generated keys. Section 4 describes the modulus classification method and its results on large real-world key sets, the practical impact and mitigation of which are discussed in Section 5. Additional analysis performed for black-box implementations on cards and a discussion of the practical impact of a faulty/biased random number generator are presented in Section 6. Finally, conclusions are offered in Section 7.

²The correct library is listed within the first three most probable groups of distinct sources identified by the classification algorithm.

2 RSA key pairs

To use the RSA algorithm, one must generate a key:

1. Select two distinct large primes³ p and q .
2. Compute $n = p \cdot q$ and $\varphi(n) = (p - 1)(q - 1)$.
3. Choose a public exponent⁴ $e < \varphi(n)$ that is coprime to $\varphi(n)$.
4. Compute the private exponent d as $e^{-1} \bmod \varphi(n)$.

The pair (e, n) is the public key; either (d, n) serves as the secret private key, or (p, q) can be used ((d, n) can be calculated from (p, q, e) and vice versa).

2.1 Attacks against the RSA cryptosystem

The basic form of attack on the RSA cryptosystem is modulus factorization, which is currently computationally unfeasible or at least extremely difficult if p and q are sufficiently large (512 bits or more) and a general algorithm such as the number field sieve (NFS) or the older quadratic sieve (MPQS) is used. However, special properties of the primes enable more efficient factorization, and measures may be taken in the key pair generation process to attempt to prevent the use of such primes.

The primes used to generate the modulus should be of approximately the same size because the factorization time typically depends on the smallest factor. However, if the primes are too close in value, then they will also be close to the square root of n and *Fermat factorization* can be used to factor n efficiently [16].

Pollard's $p - 1$ method outperforms general algorithms if for one of the primes p , $p - 1$ is B -smooth (all factors are $\leq B$) for some small B (which must usually be guessed in advance). The modulus can be factored using *Williams' $p + 1$* method if $p + 1$ has no large factors [27].

Despite the existence of many special-purpose algorithms, the easiest way to factor a modulus created as the product of two randomly generated primes is usually to use the NFS algorithm. Nevertheless, using special primes may potentially thwart such factorization attacks, and some standards, such as ANSI X9.31 [28] and FIPS 186-4 [14], require the use of primes with certain properties (e.g., $p - 1$ and $p + 1$ must have at least one large factor). Other special algorithms, such as *Pollard's rho* method and the *Lenstra elliptic curve* method, are impractical for factoring a product of two large primes.

Although RSA factorization is considered to be an NP-hard problem if keys that fulfil the above conditions are used, practical attacks, often relying on a faulty random

³Generated randomly, but possibly with certain required properties, as we will see later.

⁴Usually with a low Hamming weight for faster encryption.

generator, nevertheless exist. Insufficient entropy, primarily in routers and embedded devices, leads to weak and factorizable keys [12]. A faulty card random number generator has produced weak keys for Taiwanese citizens [3], and supposedly secure cryptographic tokens have been known to produce corrupted or significantly biased keys and random streams [6].

Implementation attacks can also compromise private keys based on leakage in side channels of timing [8] or power [15]. Active attacks based on fault induction [26] or exploits aimed at message formatting [2, 5] enable the recovery of private key values. We largely excluded these classes of attacks from the scope of our analysis, focusing only on key generation.

2.2 Analysis methodology

Our purpose was to verify whether the RSA key pairs generated from software libraries and on cards provide the desired quality and security with respect to the expectations of randomness and resilience to common attacks. We attempted to identify the characteristics of the generated keys and deduce the process responsible for introducing them. The impact of the techniques used on the properties of the produced public keys was also investigated. We used the following methodology:

1. Establish the characteristics of keys generated from open-source cryptographic libraries with known implementations.
2. Gather a large number of RSA key pairs from cryptographic software libraries and cards (one million from each).
3. Compare the keys originating from open-source libraries and black-box implementations and discuss the causes of any observed similarities and differences (e.g., the distribution of the prime factors of $p - 1$).
4. Analyse the generated keys using multiple statistical techniques (e.g., calculate the distribution of the most significant bytes of the primes).

Throughout this paper, we will use the term *source* (of keys) when referring to both software libraries and cards.

2.3 Source code and literature

We examined the source codes of 19 open-source cryptographic libraries variants⁵ and match it to the relevant algorithms for primality testing, prime generation and

⁵We inspected multiple versions of libraries (though not all exhaustively) to detect code changes relevant to the key generation process. If such a change was detected, both versions were included in the analysis.

RSA key generation from standards and literature. We then examined how the different methods affected the distributions of the primes and moduli. Summary results together for all sources are available in Table 1.

2.3.1 Prime generation

Probable primes. Random numbers (or numbers from a sequence) are tested for primality using probabilistic primality (compositeness) tests. Different libraries use different combinations of the Fermat, Miller-Rabin, Solovay-Strassen and Lucas tests. None of the tests rejects prime numbers if implemented correctly; hence, they do not affect the distribution of the generated primes. GNU Crypto uses a flawed implementation of the Miller-Rabin test. As a result, it permits only Blum primes⁶. No other library generates such primes exclusively (however, some cards do).

In the *random sampling* method, large integers (candidates) are generated until a prime is found. If the candidates are chosen uniformly, the distribution is not biased (case of GNU Crypto 2.0.1, LibTomCrypt 1.17 and WolfSSL 3.9.0). An *incremental search* algorithm selects a random candidate and then increments it until a prime is found (Botan 1.11.29, Bouncy Castle 1.54, Cryptix 20050328, cryptlib 3.4.3, Crypto++ 5.6.3, FlexiProvider 1.7p7, mbedTLS 2.2.1, SunRsaSign – OpenJDK 1.8.0, OpenSSL 1.0.2g, and PGPSDK4). Primes preceded by larger “gaps” will be selected with slightly higher probability; however, this bias is not observable from the distribution of the primes.

Large random integers are likely to have some small prime divisors. Before time-consuming primality tests are performed, compositeness can be revealed through trial division with small primes or the computation of the greatest common divisor (GCD) with a product of a few hundred primes. In the case of incremental search, the sieve of Eratosthenes or a table of remainders that is updated when the candidate is incremented can be used. If implemented correctly, these efficiency improvements do not affect the distribution of the prime generator.

OpenSSL creates a table of remainders by dividing a candidate by small primes. When a composite candidate is incremented, this table is efficiently updated using only operations with small integers. Interestingly, candidates for p for which $p - 1$ is divisible by a small prime up to 17 863 (except 2) are also rejected. Such a computational step is useful to speed up the search for a *safe prime*; however, $(p - 1)/2$ is not required (as would be for *safe prime*) to be prime by the library. This strange behaviour was first reported by Mironov [20] and can be used to classify the source if the primes are known.

⁶A prime p is a Blum prime if $p \equiv 3 \pmod{4}$. When both p and q are Blum primes, the modulus n is a Blum integer $n \equiv 1 \pmod{4}$.

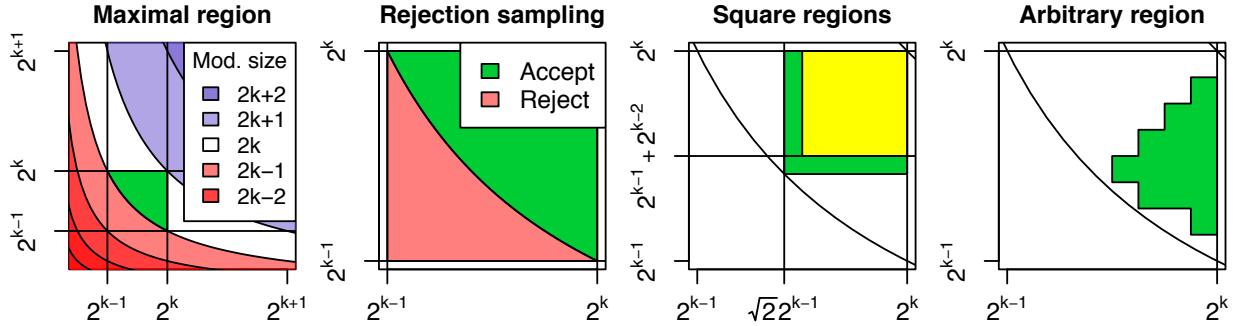


Figure 1: RSA key generation. The maximal region for the generated primes is defined by the precise length of the modulus and equal lengths of the primes. Such keys can be generated through rejection sampling. To avoid generating short moduli (which must be discarded), alternative square regions may be used. Several implementations, such as that of the NXP J2A080 card, generate primes from arbitrary combinations of square regions.

Provable primes. Primes are constructed recursively from smaller primes, such that their primality can be deduced mathematically (using Pocklington’s theorem or related facts). This process is randomized; hence, a different prime is obtained each time. An algorithm for constructing provable primes was first proposed by Maurer [18] (used by Nettle 3.2). For each prime p , $p - 1$ must have a large factor ($\geq \sqrt{p}$ for Maurer’s algorithm or $\geq \sqrt[3]{p}$ for an improved version thereof). Factors of $p + 1$ are not affected.

Strong primes. A prime p is *strong* if both $p - 1$ and $p + 1$ have a large prime factor (used by libgcrypt 1.65 in FIPS mode and by the OpenSSL 2.0.12 FIPS module). We also refer to these primes as *FIPS-compliant*, as FIPS 186-4 requires such primes for 1024-bit keys (larger keys may use probable primes). Differing definitions of strong primes are given in the literature; often, the large factor of $p - 1$ itself (minus one) should also have a large prime factor (PGPSDK4 in FIPS mode). Large random primes are not “weak” by comparison, as their prime factors are sufficiently large, with sufficient probability, to be safe from relevant attacks.

Strong primes are constructed from large prime factors. They can be generated uniformly (as in ANSI X9.31, FIPS 186-4, and IEEE 1363-2000) or with a visibly biased distribution (as in a version of Gordon’s algorithm [11] used in PGPSDK4).

2.3.2 Key generation – prime pairs

The key size is the bit length of the modulus. Typically, an algorithm generates keys of an exact bit length (the only exception being PGPSDK4 in FIPS mode). The primes are thus generated with a size equal to half of the modulus length. These two measures define the *maximal region* for RSA primes. The product of two k -bit primes is either $2k$ or $2k - 1$ bits long. There are two princi-

pal methods of solving the problem of short $(2k - 1)$ -bit moduli, as illustrated in Figure 1.

Rejection sampling. In this method, pairs of k -bit primes are generated until their product has the correct length. To produce an unbiased distribution, two new primes should be generated each time (Cryptix 20050328, FlexiProvider 1.7p7, and mbedTLS 2.2.1). If the greater prime is kept and only one new prime is generated, some bias can be observed in the resulting distribution of RSA moduli (Bouncy Castle up to version 1.53 and SunRsaSign in OpenJDK 1.8.0). If the first prime is kept (without regard to its size) and the second prime is re-generated, small moduli will be much more probable than large values (GNU Crypto 2.0.1).

“Square” regions. This technique avoids the generation of moduli of incorrect length that must be discarded by generating only larger primes such that their product has the correct length. Typically, both primes are selected from identical intervals. When the prime pairs are plotted in two dimensions, this produces a square region.

The smallest k -bit numbers that produce a $2k$ -bit modulus are close to $\sqrt{2} \cdot 2^{k-1}$. Because random numbers can easily be uniformly generated from intervals bounded by powers of two, the distribution must be additionally transformed to fit such an interval. We refer to prime pairs generated from the interval $[\sqrt{2} \cdot 2^{k-1}, 2^k - 1]$ as being generated from the *maximal square region* (Bouncy Castle since version 1.54, Crypto++ 5.6.3, and the Microsoft cryptography providers used in CryptoAPI, CNG and .NET). Crypto++ approximates this interval by generating the most significant byte of primes from 182 to 255.

A more *practical square region*, which works well for candidates generated uniformly from intervals bounded by powers of two, is achieved by fixing the two most significant bits of a candidate to 11₂ (Botan 1.11.29, cryptlib

3.4.3, libgcrypt 1.6.5, LibTomCrypt 1.17, OpenSSL 1.0.2g, PGPSDK4, and WolfSSL 3.9.0). Additionally, the provable primes generated in Nettle 3.2 and the strong primes generated in libgcrypt 1.6.5 (in FIPS mode) and in the OpenSSL 2.0.12 FIPS module are produced from this region.

2.4 Analysis of black-box implementations

To obtain representative results of the key generation procedures used in cards (for which we could not inspect the source codes), we investigated 16 different types of cards from 6 different established card manufacturers ($2 \times$ Gemalto, $6 \times$ NXP, $1 \times$ Infineon, $3 \times$ Giesecke & Devrient (G&D), $2 \times$ Feitian and $2 \times$ Oberthur) developed using the widely used JavaCard platform. The key pair generation process itself is implemented at a lower level, with JavaCard merely providing an interface for calling relevant methods. For each type of card (e.g., NXP J2D081), three physical cards were tested to detect any potential differences among physical cards of the same type (throughout the entire analysis, no such difference was ever detected). Each card was programmed with an application enabling the generation and export of an RSA key pair (using the `KeyPair.generateKey()` method) and truly random data (using the `RandomData.generate()` method).

We focused primarily on the analysis of RSA keys of three different lengths – 512, 1024 and 2048 bits. Each card was repeatedly asked to generate new RSA 512-bit key pairs until one million key pairs had been generated or the card stopped responding. The time required to create these key pairs was measured, and both the public (the modulus n and the exponent e) and private (the primes p and q and the private exponent d) components were exported from the card for subsequent analyses. No card reset was performed between key pair generations. In the ideal case, three times one million key pairs were extracted for every card type. The same process was repeated for RSA key pairs with 1024-bit moduli but for only 50 000 key pairs, as the key generation process takes progressively longer for longer keys. The patterns observed from the analyses performed on the 512-bit keys was used to verify the key set with longer keys⁷.

Surprisingly, we found substantial differences in the intervals from which primes were chosen. In some cases, non-uniform distributions of the primes hinted that the prime generation algorithms are also different to those used in the software libraries. Several methods adopted in software libraries, such as incremental search, seem to be suitable even for limited-resource systems. This

⁷For example, one can quickly verify whether a smaller number of factorized values of $p - 1$ from 1024-bit RSA keys fit the distribution extrapolated from 512-bit keys.

argument is supported by a patent application [21] by G&D, one of the manufacturers of the examined cards. All tested cards from this manufacturer produced Blum integers, as described in the patent, and these integers were distributed uniformly, as expected from the incremental search method.

A duration of approximately 2-3 weeks was typically required to generate one million key pairs from a single card, and we used up to 20 card readers gathering keys in parallel. Not all cards were able to generate all required keys or random data, stopping with a non-specific error (0x6F00) or becoming permanently non-responsive after a certain period. In total, we gathered more than 30 million card-generated RSA key pairs⁸. Power consumption traces were captured for a small number of instances of the key pair generation process.

In addition, 100 MB streams of truly random data were extracted from each card for tests of statistical randomness. When a problem was detected (i.e., the data failed one or more statistical tests), a 1 GB stream was generated for fine-grained verification tests.

3 Analysis of the generated RSA key pairs

The key pairs extracted from both the software libraries and the cards were examined using a similar set of analytical techniques. The goal was to identify sources with the same behaviour, investigate the impact on the public key values and infer the probable key generation algorithm used based on similarities and differences in the observed properties.

3.1 Distributions of the primes

To visualize the regions from which pairs of primes were chosen, we plotted the most significant byte (MSB) of each prime on a heat map. It is possible to observe the *intervals for prime generation*, as discussed in Section 2.3.

Figure 2 shows a small subset of the observed non-uniform distributions. Surprisingly, the MSB patterns were significantly different for the cards and the software implementations. The patterns were identical among different physical cards of the same type and were also shared between some (but not all) types of cards from the same manufacturer (probably because of a shared code base). We did not encounter any library that produced outputs comparable to those of the first two cards from the examples shown in Figure 2. The third example could be reproduced by generating primes alternately and uniformly from 14 different regions, each characterized by a pattern in the top four bits of the primes. By comparison, it was rarer for a bias to be introduced by a library.

⁸The entire dataset is available for further research at [31].

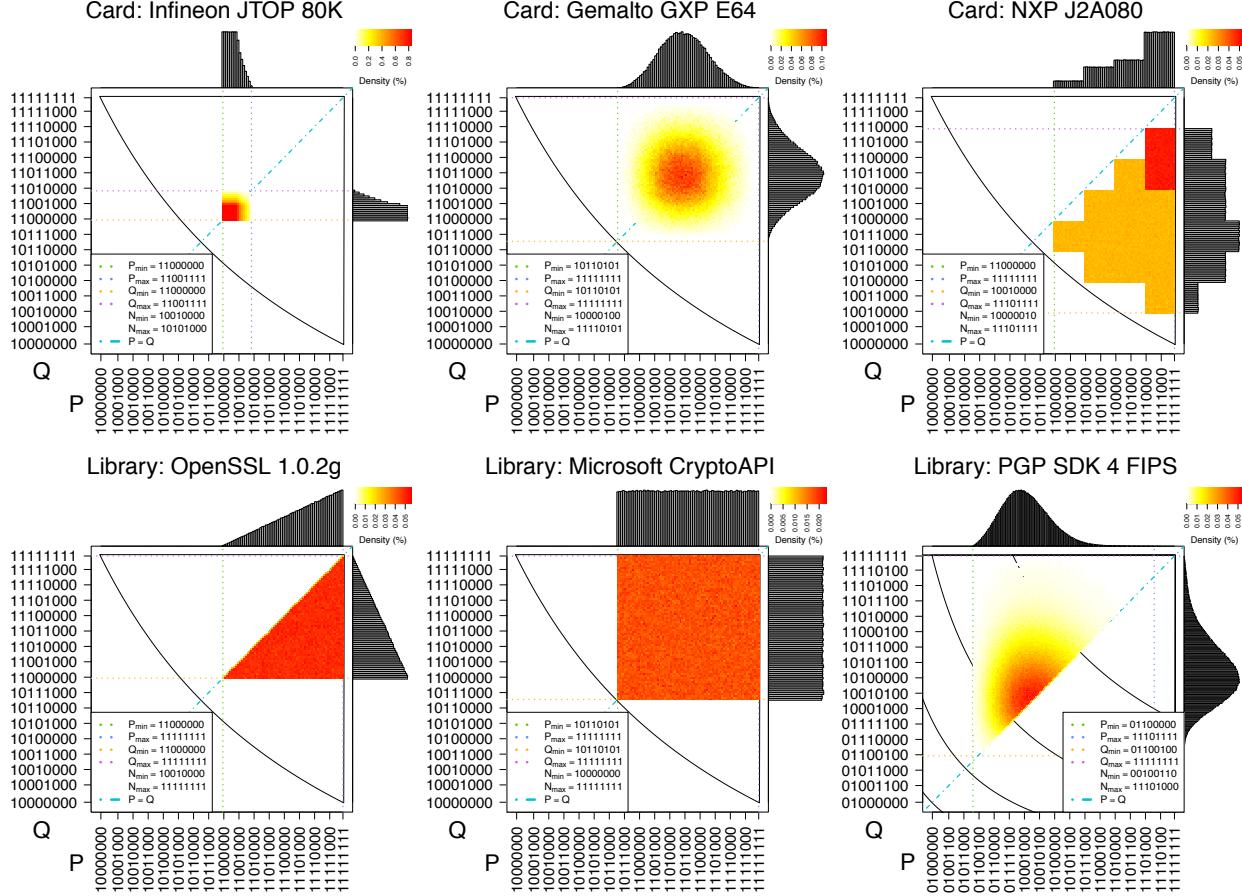


Figure 2: Example distributions of the most significant byte of the prime p and the corresponding prime q from 8 million RSA key pairs generated by three software libraries and three types of cards. The histograms on the top and the side of the graph represent the marginal distributions of p and q , respectively. The colour scheme expresses the likelihood that primes of a randomly generated key will have specific high order bytes, ranging from white (not likely) over orange to red (more likely). For distributions of all sources from the dataset, see our technical report [25].

The relation between the values of p and q reveals additional conditions placed on the primes, such as a minimal size of the difference $p - q$ (PGPSDK4, NXP J2D081, and NXP J2E145G).

It is possible to verify whether small factors of $p - 1$ are being avoided (e.g., OpenSSL or NXP J2D081) or whether the primes generally do not exhibit same distribution as randomly generated numbers (Infineon JT0P 80K) by computing the distributions of the primes, modulo small primes. It follows from Dirichlet's theorem that the remainders should be distributed uniformly among the $\phi(n)$ congruence classes in \mathbb{Z}_n^* [19, Fact 4.2].

The patterns observed for the 512-bit keys were found to be identical to those for the stronger keys of 1024 and 2048 bits. For the software implementations, we checked the source codes to confirm that there were no differences in the algorithms used to generate keys of different lengths. For the cards, we assume the same and gen-

eralize the results obtained for 512-bit RSA keys to the longer (and much more common) keys.

3.2 Distributions of the moduli

The MSB of a modulus is directly dependent on the MSBs of the corresponding primes p and q . As seen in Figure 3, if an observable pattern exists in the distributions of the MSBs of primes p and q , a noticeable pattern also appears in the MSB of the modulus. The preservation of shared patterns was observed for all tested types of software libraries and cards. The algorithm used for prime pair selection can often be observed from the distribution of the moduli. If a source uses an atypical algorithm, it is possible to detect it with greater precision, even if we do not know the actual method used.

Non-randomness with respect to small factors of $p - 1$ can also be observed from the modulus, especially for

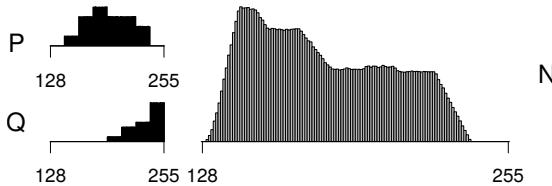


Figure 3: The visible preservation of the MSB distributions of the primes p and q in the MSB distribution of the modulus $n = p \cdot q$. This example is from the NXP J2A081 card.

small divisors. Whereas random primes are equiprobably congruent to 1 and 2 modulo 3, OpenSSL primes are always congruent to 2. As a result, an OpenSSL modulus is always congruent to 1 modulo 3. This property is progressively more difficult to detect for larger prime divisors. The moduli are more probably congruent to 1 modulo all small primes, which are avoided from $p - 1$ by OpenSSL. However, the bias is barely noticeable for prime factors of 19 and more, even in an analysis of a million keys. OpenSSL primes are congruent to 1 modulo 5 with probability 1/3 (as opposed to 1/4 for random primes), to 1 modulo 7 with probability 1/5 (as opposed to 1/6), and to 1 modulo 11 with probability 1/9 (as opposed to 1/10). For the practical classification of only a few keys (see Section 4), we use only the remainder of division by 3.

The use of *Blum integers* can also be detected from the moduli with a high precision, as random moduli are equiprobably congruent to 1 and 3 modulo 4, whereas Blum integers are always congruent to 1 modulo 4. The probability that k random moduli will be Blum integers is 2^{-k} .

Neither libraries nor cards attempt to achieve a uniform distribution of moduli. Existing algorithms [13, 17] have the disadvantage that sometimes a prime will be one bit larger than half of the modulus length. All sources sacrifice uniformity in the most significant bits of the modulus to benefit from more efficient methods of prime and key generation.

We verified that the distribution of the other bytes of the moduli is otherwise uniform. The second least significant bit is biased in the case of Blum integers. Sources that use the same algorithm are not mutually distinguishable from the distributions of their moduli.

3.3 Factorization of $p - 1$ and $p + 1$

It is possible to verify whether *strong primes* are being used. Most algorithms generate strong primes from uniform distributions (ANSI X9.31, FIPS 186-4, IEEE 1363, OpenSSL FIPS, libgcrypt FIPS, Microsoft and Gemalto GCX4 72K), matching the distribution of ran-

dom primes, although PGPSDK4 FIPS produces a highly atypical distribution of primes and moduli, such that this source can be detected even from public keys. Hence, we were obliged to search for the sizes of the prime factors of $p - 1$ and $p + 1$ directly⁹ by factoring them using the YAFU software package [29]. We then extended the results obtained for 512-bit keys to the primes of 1024-bit key pairs (though based on fewer factorized values because of the longer factorization time). Finally, we extrapolated the results to keys of 2048 bits and longer based on the known patterns for shorter keys.

As confirmed by the source code, large factors of $p \pm 1$ generated in OpenSSL FIPS and by libgcrypt in FIPS mode always have 101 bits; this value is hardcoded. PGPSDK4 in FIPS mode also generates prime factors of fixed length; however, their size depends on the size of the prime.

Additionally, we detected strong primes in some of our black-box sources. Gemalto GCX4 72K generates strong primes uniformly, but the large prime factors always have 101 bits. The strong primes of Gemalto GXP E64, which have 112 bits, are not drawn from a uniform distribution. The libraries that use Microsoft cryptography providers (CryptoAPI, CNG, and .NET) produce prime factors of randomized length, ranging from 101 bits to 120 bits, as required by ANSI X9.31.

For large primes, $p \pm 1$ has a large prime factor with high probability. A random integer p will not have a factor larger than $p^{1/u}$ with a probability of approximately u^{-u} [19]. Approximately 10% of 256-bit primes do not have factors larger than 100 bits, but 512-bit keys are not widely used. For 512-bit primes, the probability is less than 0.05%. Therefore, the requirement of a large factor does not seem to considerably decrease the number of possible primes. However, many sources construct strong primes with factors of exact length (e.g., 101 bits). Using the approximation of the prime-counting function $\pi(n) \approx \frac{n}{\ln(n)}$ [19], we estimate that the interval required by ANSI X9.31 (prime factors from 101 to 120 bits) contains approximately 2^{20} times more primes than the number of 101-bit primes. Hence, there is a loss of entropy when strong primes are generated in this way, although we are not aware of an attack that would exploit this fact. For every choice of an auxiliary prime, 2^{93} possible values are considered instead of 2^{113} , which implies the loss of almost 20 bits of entropy. If the primes are to be (p^-, p^+) -safe, then 2 auxiliary primes must be generated. Because we require two primes p and q for every RSA key, we double the estimated loss of entropy compared with ANSI-compliant keys to 80 bits for 1024-bit keys.

When $p - 1$ is guaranteed to have a large prime factor but $p + 1$ is not, the source is most likely using *provable*

⁹By $p - 1$, we always refer to both $p - 1$ and $q - 1$, as we found no relevant difference between p and q in the factorization results.

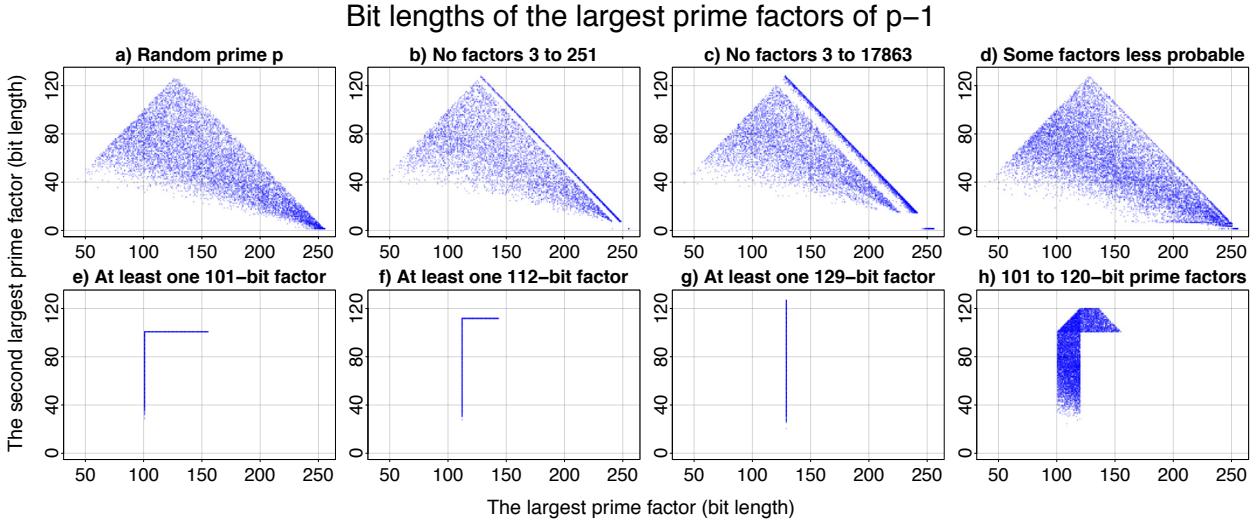


Figure 4: Scatter graphs with all combinations of two biggest factors of $p - 1$ for 512-bit RSA. The tested sources fall into following categories: **a)** Botan 1.11.29, Bouncy Castle 1.53 & 1.54, Cryptix JCE 20050328, cryptlib 3.4.3, Crypto++ 5.6.3, FlexiProvider 1.7p7, GNU Crypto 2.0.1, (GPG) libgcrypt 1.6.5, LibTomCrypt 1.17, mbedTLS 2.2.1, PGPSDK4, SunRsaSign (OpenJDK 1.8), G&D SmartCafe 3.2, Feitian JavaCOS A22, Feitian JavaCOS A40, NXP J2A080, NXP J2A081, NXP J3A081, NXP JCOP 41 V2.2.1, Oberthur Cosmo Dual 72K; **b)** NXP J2D081, NXP J2E145G; **c)** OpenSSL 1.0.2g; **d)** Infineon JT0P 80K; **e)** (GPG) libgcrypt 1.6.5 FIPS, OpenSSL FIPS 2.0.12, Gemalto GCX4 72K; **f)** Gemalto GXP E64; **g)** Nettle 3.2; **h)** MS CNG, MS CryptoAPI, MS .NET.

primes, as in the case of the Nettle library. Techniques for generating provable primes construct p using a large prime factor of $p - 1$ (at least \sqrt{p} for Maurer's algorithm or $\sqrt[3]{p}$ for an improved version thereof). The size of the prime factors of $p + 1$ is not affected by Maurer's algorithm.

Factorization also completes the picture with regard to the avoidance of small factors in $p - 1$. Sources that avoid small factors in $p - 1$ achieve a smaller number of factors on average (and therefore also a higher average length of the largest factor). No small factors are present in keys from NXP J2D081 and J2E145G (values from 3 to 251 are avoided), from OpenSSL (values from 3 to 17 863 are avoided) and from G&D Smartcafe 4.x and G&D Smartcafe 6.0 (values 3 and 5 are avoided). Small factors in $p + 1$ are not avoided by any source.

Concerning the *distribution of factors*, most of the software libraries (14) and card types (8) yield distributions comparable to that of randomly generated numbers of a given length (see Figure 4). The Infineon JT0P 80K card produces significantly more small factors than usual (compared with both random numbers and other sources). This decreases the probability of having a large factor.

We estimated the percentage of 512-bit RSA keys that are susceptible to Pollard $p - 1$ factorization within 2^{80} operations. This percentage ranges from 0% (FIPS-compliant sources) to 4.35% (Infineon JCOP 80K), with

an average of 3.38%. Although the NFS algorithm would still be faster in most cases of keys of 512 bits and larger, we found a card-generated key (with a small maximal factor of $p - 1$) that was factorized via Pollard $p - 1$ method in 19 minutes, whereas the NFS algorithm would require more than 2 000 CPU hours. Note that for 1024-bit keys, the probability of such a key being produced is negligible.

3.4 Sanity check

Based on the exported private and public components of the generated RSA keys obtained from all sources, we can summarize their basic properties as follows (see also Table 1):

- All values p and q are primes and are not close enough for Fermat factorization to be practical.
- All card-generated keys use a public exponent equal to 0x10001 (65 537), and all software libraries either use this value as the default or support a user-supplied exponent.
- Most modulus values are of an exactly required length (e.g., 1024 bits). The only exception is PGPSDK4 in FIPS mode, which also generates moduli that are shorter than the specified length by one or two bits.

- Neither libraries nor cards ensure that p is a safe prime ($p = 2 \cdot q + 1$, where q is also prime).
- Some sources construct strong primes according to the stricter definition or at least comply with the requirements defined in the FIPS 186-4 and ANSI X9.31 standards, such that $p - 1$ and $p + 1$ both have a large prime factor. Other libraries are not FIPS-compliant; however, keys of 1024 bits and larger resist $p - 1$ and $p + 1$ attacks for practical values of the smoothness bound.
- Some libraries (5) and most card types (12) order the primes such that $p > q$, which seems to be a convention for CRT RSA keys. PGPSDK4 (in both regular and FIPS modes) and libgcrypt (used by GnuPG) in both modes order the primes in the opposite manner, $q > p$. In some sources, the ordering is a side effect of the primes having fixed (and different) most significant bits (e.g., 4 bits of p and q are fixed to 1111 and 1001, respectively, by all G&D cards).
- All generated primes were unique for all libraries and all types of cards except one (Oberthur Cosmo Dual 72K).
- All G&D and NXP cards, the Oberthur Cosmo Dual 72K card and the GNU Crypto library generate Blum integers. As seen from a bug in the implementation of the Miller-Rabin test in GNU Crypto, a simpler version of the test suffices for testing Blum primes. However, we hypothesize that the card manufacturers have a different motivation for using such primes.

4 Key source detection

The distinct distributions of specific bits of primes and moduli enable probabilistic estimation of the source library or card from which a given public RSA key was generated. Intuitively, classification works as follows: 1) Bits of moduli known to carry bias are identified with additional bits derived from the modulus value (a *mask*, 6 + 3 bits in our method). 2) The frequencies of all possible *mask* combinations (2^9) for a given source in the learning set are computed. 3) For classification of an unknown public key, the bits selected by the *mask* are extracted as a particular value v . The source with the highest computed frequency of value v (step 2) is identified as the most probable source. When more keys from the same source are available (multiple values v_i), a higher classification accuracy can be achieved through element-wise multiplication of the probabilities of the individual keys.

We first describe the creation of a classification matrix and report the classification success rate as evaluated on our test set [31]. Later, classification is applied to three real-world datasets: the IPv4 HTTPS handshakes set [9], Certificate Transparency set [10] and the PGP key set [30].

4.1 The classification process

The classification process is reasonably straightforward. For the full details of the algorithm, please refer to our technical report [25].

1. All modulus bits identified through previous analysis as non-uniform for at least one source are included in a *mask*. We included the $2^{nd} - 7^{th}$ most significant bits influenced by the prime manipulations described in Section 3.1, the second least significant bit (which is zero for sources that use Blum integers), the result of the modulus modulo 3 (which is influenced by the avoidance of factor 3) and the overall modulus length (which indicates whether an exact length is enforced).
2. A large number of keys (learning set) from known generating sources are used to create a *classification matrix*. For every possible *mask* value (of which there are 2^9 in our case) and every source, the relative frequency of the given mask value in the learning set for the given source is computed.
3. During the classification phase for key K with modulus m , the value v obtained after the application of *mask* to modulus m is extracted. The row (*probability vector*) of the *classification matrix* that corresponds to the value v contains, as its i^{th} element, the probability of K being produced by source i .
4. When a batch of multiple keys that are known to have been produced by the same (unknown) source is classified, the *probability vectors* for every key obtained in step 3 are multiplied element-wise and normalized to obtain the source probabilities p_b for the entire batch, and the source with the highest probability is selected.

Note that the described algorithm cannot distinguish between sources with very similar characteristics, e.g., between the NXP J2D081 and NXP J2E145G cards, which likely share the same implementation. For this reason, if two sources have the same or very similar profiles, they are placed in the same *group*. Figure 5 shows the clustering and (dis-)similarity of all sources considered in this study. If the particular source of one or more key(s) is missing from our analysis (relevant for the classification

Source	Version	Classification group	Prime search method	Prime pair selection	Blum integers	Small factors of $p - 1$	Large factor of $p - 1$	Large factor of $p + 1$	$ p - q $ check	$ d $ check	Notes
Open-source libraries											
Botan	1.11.29	XI	Incr.	11_2	×	✓	×	×	×	×	
Bouncy Castle	1.53	VIII	Incr.	RS	×	✓	×	×	✓	✓	Rejection sampling is less biased
Bouncy Castle	1.54	X	Incr.	$\sqrt{2}$	×	✓	×	×	✓	✓	Checks Hamming weight of the modulus
Cryptix JCE	20050328	VIII	Incr.	RS	×	✓	×	×	×	×	Rejection sampling is not biased
cryptlib	3.4.3	XI	Incr.	11_2	×	✓	×	×	✓	✓	
Crypto++	5.6.3	X	Incr.	$\sqrt{2}$	×	✓	×	×	×	×	$255 \geq \text{MSB of prime} \geq 182 = \lceil \sqrt{2} \cdot 128 \rceil$
FlexiProvider	1.7p7	VIII	Incr.	RS	×	✓	×	×	×	×	Rejection sampling is not biased
GNU Crypto	2.0.1	II	Rand.	RS	✓	✓	×	×	×	×	Rejection sampling is more biased
GPG Libgcrypt	1.6.5	XI	Incr.	11_2	×	✓	×	×	×	×	Used by GnuPG 2.0.30
GPG Libgcrypt	1.6.5 FIPS mode	XI	FIPS	11_2	×	✓	✓	✓	✓	✓	101-bit prime factors of $p \pm 1$
LibTomCrypt	1.17	XI	Rand.	11_2	×	✓	×	×	×	×	
mbedTLS	2.2.1	VIII	Incr.	RS	×	✓	✓	×	×	×	Rejection sampling is not biased
Nettle	3.2	XI	Maurer	11_2	×	✓	✓	×	×	×	Prime factor of $p - 1$ has $(n /4 + 1)$ bits
OpenSSL	1.0.2g	V	Incr.	11_2	×	×	×	×	×	×	No prime factors 3 to 17 863 in $p - 1$
OpenSSL FIPS	2.0.12	XI	FIPS	11_2	×	✓	✓	✓	✓	✓	101-bit prime factors of $p \pm 1$
PGP SDK 4.x	PGP Desktop 10.0.1	XI	Incr.	11_2	×	✓	×	×	✓	×	p and q differ in their top 6 bits
PGP SDK 4.x	FIPS mode	IV	PGP	11_2	×	✓	✓	✓	✓	✓	Prime factors of $p \pm 1$ have $(n /4 - 32)$ bits
SunRsaSign Provider	OpenJDK 1.8	VIII	Incr.	RS	×	✓	×	×	×	×	Rejection sampling is less biased
WolfSSL	3.9.0	XI	Rand.	11_2	×	✓	×	×	×	×	
Black-box implementations											
Microsoft CNG	Windows 10	X	FIPS	$\sqrt{2}$	×	✓	✓	✓	?	?	Prime factors of $p \pm 1$ have 101 to 120 bits
Microsoft CryptoAPI	Windows 10	X	FIPS	$\sqrt{2}$	×	✓	✓	✓	?	?	Prime factors of $p \pm 1$ have 101 to 120 bits
Microsoft .NET	Windows 10	X	FIPS	$\sqrt{2}$	×	✓	✓	✓	?	?	Prime factors of $p \pm 1$ have 101 to 120 bits
Smartcards											
Feitian JavaCOS A22		XI	Incr./Rand.	11_2	×	✓	×	×	?	?	
Feitian JavaCOS A40		XI	Incr./Rand.	11_2	×	✓	×	×	?	?	
G&D SmartCafe 3.2		XIII	Incr./Rand.	$FX \times 9X$	✓	✓	×	×	✓*	?	*Size of $ p - q $ guaranteed by prime intervals
G&D SmartCafe 4.x		I	Incr./Rand.	$FX \times 9X$	✓	×	×	×	✓*	?	No prime factors 3 and 5 in $p - 1$
G&D SmartCafe 6.0		I	Incr./Rand.	$FX \times 9X$	✓	×	×	×	✓*	?	No prime factors 3 and 5 in $p - 1$
Gemalto GCX4 72K		XI	FIPS	11_2	×	✓	✓	✓	?	?	101-bit prime factors of $p \pm 1$
Gemalto GXP E64		IX	Gem.	Gem.	×	✓	✓	✓	?	?	112-bit prime factors of $p \pm 1$
Infineon JT0P 80K		XII	Inf.	Inf.	×	✓	×	×	?	?	
NXP J2A080		VII	Incr./Rand.	NXP	✓	✓	✓	✓	?	?	
NXP J2A081		VII	Incr./Rand.	NXP	✓	✓	✓	✓	?	?	
NXP J2D081		III	Incr./Rand.	RS	✓	×	×	×	✓	?	No prime factors 3 to 251 in $p - 1$
NXP J2E145G		III	Incr./Rand.	RS	✓	×	×	×	✓	?	No prime factors 3 to 251 in $p - 1$
NXP J3A081		VII	Incr./Rand.	NXP	✓	✓	✓	✓	?	?	
NXP JCOP 41 V2.2.1		VII	Incr./Rand.	NXP	✓	✓	✓	✓	?	?	
Oberthur Cosmo Dual 72K		VI	Incr.	11_2	✓	✓	✓	✓	✓	?	
Oberthur Cosmo 64		XI	Incr./Rand.	11_2	×	✓	?	?	?	?	512-bit keys not supported

Table 1: Comparison of cryptographic libraries and smartcards. The algorithms are explained in Section 2.3. **Prime search method:** incremental search (Incr.); random sampling (Rand.); FIPS 186-4 Appendix B.3.6 or equivalent algorithm for strong primes (FIPS); Maurer’s algorithm for provable primes (Maurer); PGP strong primes (PGP); Gemalto strong primes (Gem.); Infineon algorithm (Inf.); unknown prime generator with almost uniform distribution, possibly incremental or random search (Incr./Rand.). **Prime pair selection:** practical square region (11_2); rejection sampling (RS); maximal square region ($\sqrt{2}$); the primes p and q have a fixed pattern in their top four bits, 1111_2 and 1001_2 , respectively ($FX \times 9X$); Gemalto non-uniform strong primes (Gem.); Infineon algorithm (Inf.); NXP regions – 14 distinct square regions characterized by patterns in the top four bits of p and q (NXP). **Blum integers:** the modulus n is always a Blum integer $n \equiv 1 \pmod{4}$ (✓); the modulus is $n \equiv 1 \pmod{4}$ and $n \equiv 3 \pmod{4}$ with equal probability (×). **Small factors of $p - 1$:** $p - 1$ contains small prime factors (✓); some prime factors are avoided in $p - 1$ (×). **Large factors of $p - 1$:** $p - 1$ is guaranteed to have a large prime factor – provable and strong primes (✓); size of the prime factors of $p - 1$ is random (×). **Large factors of $p + 1$:** similar as for $p - 1$, typically strong primes are (✓); random and provable primes are (×). **$|p - q|$ check:** p and q differ somewhere in their top bits (✓); the property is not guaranteed (×); the check may be performed, but the negative case occurs with a negligible probability (?). **$|d|$ check:** sufficient bit length of the private exponent d is guaranteed (✓); not guaranteed (×); possibly guaranteed, but not detectable (?).

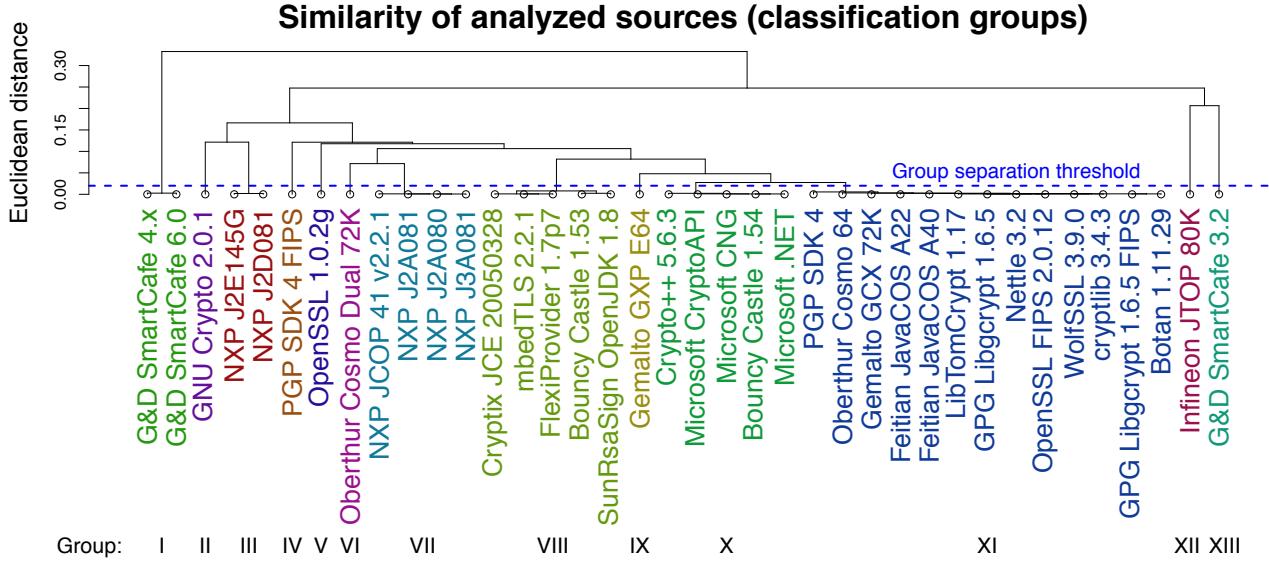


Figure 5: Clustering of all inspected sources based on the 9 bits of the mask. The separation line shows which sources were put by us into the same classification category. Finer separation is still possible (e.g., SunRsaSign vs mbedTLS), but the number of the keys from same source needs to be high enough to distinguish these very similar sources.

# keys in batch	Top 1 match					Top 2 match					Top 3 match				
	1	2	5	10	100	1	2	5	10	100	1	2	5	10	100
Group I	95.39%	98.42%	99.38%	99.75%	100.00%	98.41%	99.57%	99.92%	100.00%	100.00%	98.41%	99.84%	100.00%	100.00%	100.00%
Group II	17.75%	32.50%	58.00%	69.50%	98.00%	35.58%	60.88%	84.15%	93.80%	100.00%	42.85%	71.58%	91.45%	98.40%	100.00%
Group III	45.36%	72.28%	93.17%	98.55%	100.00%	54.34%	78.31%	95.23%	99.35%	100.00%	82.45%	94.59%	99.25%	99.90%	100.00%
Group IV	90.14%	97.58%	99.80%	100.00%	100.00%	92.22%	98.14%	99.90%	100.00%	100.00%	94.42%	99.02%	100.00%	100.00%	100.00%
Group V	63.38%	81.04%	97.50%	99.60%	100.00%	84.14%	90.88%	99.25%	99.90%	100.00%	90.01%	96.62%	99.95%	100.00%	100.00%
Group VI	54.68%	69.22%	88.45%	94.60%	100.00%	80.31%	89.70%	97.90%	99.80%	100.00%	90.40%	96.34%	99.55%	100.00%	100.00%
Group VII	7.58%	31.69%	64.21%	82.35%	99.75%	32.67%	69.48%	95.33%	98.60%	100.00%	63.99%	88.70%	98.89%	99.70%	100.00%
Group VIII	15.65%	40.30%	68.46%	76.60%	85.20%	30.29%	52.81%	79.54%	92.38%	100.00%	39.32%	66.45%	90.34%	97.92%	100.00%
Group IX	22.22%	45.12%	76.35%	83.00%	83.00%	54.57%	71.86%	85.25%	86.80%	88.00%	61.77%	81.96%	94.35%	95.00%	99.00%
Group X	0.63%	6.33%	27.42%	42.74%	69.60%	15.05%	43.84%	78.83%	84.62%	91.00%	41.46%	70.54%	96.78%	99.88%	100.00%
Group XI	11.77%	28.40%	55.56%	65.28%	77.69%	29.94%	56.09%	86.43%	96.19%	100.00%	55.35%	78.48%	97.04%	99.77%	100.00%
Group XII	60.36%	79.56%	97.20%	99.40%	100.00%	82.96%	93.58%	99.60%	99.90%	100.00%	94.48%	97.62%	99.75%	100.00%	100.00%
Group XIII	39.56%	70.32%	96.20%	99.70%	100.00%	84.52%	95.54%	99.85%	100.00%	100.00%	95.22%	99.00%	99.95%	100.00%	100.00%
Average	40.34%	57.90%	78.59%	85.47%	93.33%	59.62%	76.98%	92.40%	96.26%	98.38%	73.09%	87.75%	97.48%	99.27%	99.92%

Table 2: The classification success rate of 13 groups created from all 38 analyzed sources using test set with same prior probability of sources (see Figure 5 for libraries and cards in particular group). Columns corresponds to different number of keys (1, 2, 5, 10 and 100) classified together from same (unknown) source.

of real-world datasets), any such key will be misclassified as belonging to a group with a similar mask *probability vector*.

Both the construction of the classification matrix and the actual classification are then performed using these *groups* instead of the original single sources. The observed similarities split the examined sources into 13 different groups (labelled I to XIII and listed in Figure 5). The resulting classification matrix¹⁰ has dimensions of 13×512 .

¹⁰Because of its large size, the resulting matrix is available in our technical report [25] and at <http://crcs.cz/papers/usenix2016>.

4.1.1 Evaluation of the classification accuracy

To evaluate the classification success of our method, we randomly selected 10 000 keys from the collected dataset (that were not used to construct the classification matrix) for every source, thereby endowing the test set with equal prior probability for every source.

A single organization may use the same source library to generate multiple keys for its web servers. The classification accuracy was therefore evaluated not only for one key (step 3 of the algorithm) but also for five, ten and one hundred keys (step 4) originating from the same (unknown) source. We evaluated not only the ability to

achieve the “best match” with the correct source group but also the ability to identify the correct source group within the top two and top three most probable matches (top- n match).

As shown in Table 2, the average accuracy on the test set of the most probable source group was over 40% for single keys and improved to greater than 93% when we used batches of 100 keys from the same source for classification. When 10 keys from the same source were classified in a batch, the most probable classified group was correct in more than 85% of cases and was almost always (99%) included in the top three most probable sources.

A significant variability in classification success was observed among the different groups. Groups I (G&D cards) and IV (PGPSDK4 FIPS) could be correctly identified from even a single key because of their distinct distributions of possible mask values. By contrast, group X (Microsoft providers) was frequently misclassified when only a single key was used because of the wider range of possible mask values, resulting in a lower probability of each individual mask value.

We conclude that our classification method is moderately successful even for a single key and very accurate when a batch of at least 10 keys from the same source is classified simultaneously.

Further leakage in other bits of public moduli might be found by applying machine learning methods to the learning set, potentially leading to an improvement of the classification accuracy. Moreover, although we have already tested a wide range of software libraries and cards, more sources could also be incorporated, such as additional commercial libraries, various hardware security modules and additional types of cards and security tokens.

4.2 Classifying real-world keys

One can attempt to classify keys from suitable public datasets using the described method. However, the classification of keys observed in the real world may differ from the classification scenario evaluated above in two respects:

1. The prior probabilities of real-world sources can differ significantly (e.g., OpenSSL is a more probable source for TLS keys than is any card), and the resulting posterior probabilities from the classification matrix will then also be different.
2. Our classification matrix does not include all existing sources (e.g., we have not tested high-speed hardware security modules), and such sources will therefore always be misclassified.

The classification success rate can be significantly improved if the prior distribution of possible sources can be

estimated. Such an estimate can be performed based on meta information such as statistics concerning the popularity of various software libraries or sales figures for a particular card model. Note that the prior distributions may also significantly differ for different application areas, e.g., PGP keys are generated by a narrower set of libraries and devices than are TLS keys. In this work, we did *not* perform any prior probability estimations.

4.2.1 Sources of Internet TLS keys

We used IPv4 HTTPS handshakes collected from the Internet-Wide Scan Data Repository [9] as our source of real-world TLS keys. The complete set contains approximately 50 million handshakes; the relevant subset, which consists of handshakes using RSA keys with a public exponent of 65 537, contains 33.5M handshakes. This set reduces to 10.7M unique keys based on the modulus values. The keys in this set can be further divided into *batches* with the same subject and issue date (as extracted from their certificates), where the same underlying library is assumed to be responsible for the generation of all keys in a given *batch*. As the classification accuracy improves with the inclusion of more keys in a *batch*, we obtained classification results separately for batches consisting of a single key only (users with a single HTTPS server), 2-9 keys, 10-99 keys (users with a moderate number of servers) and 100 and more keys (users with a large number of servers).

Intuitively, batches with 100+ keys will yield very accurate classification results but will capture only the behaviour of users with a large number of HTTPS servers. Conversely, batches consisting of only a single key will result in low accuracy but can capture the behaviours of different types of users.

The frequency of a given source in a dataset (for a particular range of batch sizes) is computed as follows: 1) The classification probability vector p_b for a given batch is computed according to the algorithm from Section 4.1. 2) The element-wise sum of $p_b \cdot n_b$ over all batches b (weighted by the actual number of keys n_b in the given batch) is computed and normalized to obtain the relative proportion vector, which can be found as a row in Table 3.

As shown in Section 4.1.1, a batch of 10 keys originating from the same source should provide an average classification accuracy of greater than 85% – sufficiently high to enable reasonable conclusions to be drawn regarding the observed distribution. Using batches of 10-99 keys, the highest proportion of keys generated for TLS IPv4 (82%) were classified as belonging to group V, which contains a single library – OpenSSL. This proportion increased to almost 90% for batches with 100+ keys. The second largest proportion of these keys (ap-

Dataset (size of included batches)	#keys	Group of sources												
		I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII
Multiple keys classified in single batch, likely accurate results (see discussion in Section 4.1.1)														
TLS IPv4 (10-99 keys) [9]	518K	-	0.00%	-	0.01%	82.84%	-	-	1.09%	0.28%	10.18%	5.61%	-	-
TLS IPv4 (100+ keys) [9]	973K	-	-	-	0.01%	89.92%	-	-	4.68%	0.00%	3.46%	1.93%	-	-
Cert. Transparency (10-99 keys) [10]	23K	-	0.00%	-	0.07%	26.14%	-	-	6.90%	2.79%	47.70%	16.41%	-	-
PGP keyset (10-99 keys) [30]	1.7K	-	-	-	6.87%	11.95%	-	-	36.11%	2.09%	5.73%	37.25%	-	-
Classification based on batches with 2-9 keys only, likely lower accuracy results														
TLS IPv4 (2-9 keys) [9]	237K	0.02%	0.79%	2.06%	0.11%	54.14%	3.26%	1.73%	7.03%	7.98%	11.34%	11.17%	0.36%	0.05%
Cert. Transparency (2-9 keys) [10]	794K	0.03%	1.12%	3.21%	0.14%	43.89%	5.03%	2.64%	6.59%	10.52%	12.10%	14.18%	0.49%	0.06%
PGP keyset (2-9 keys) [30]	83K	0.02%	1.47%	1.40%	2.07%	14.36%	7.90%	3.91%	7.74%	16.10%	18.80%	25.86%	0.35%	0.03%
Classification based on single key only, likely low accuracy results														
TLS IPv4 (1 key) [9]	8.8M	0.98%	4.02%	6.47%	1.94%	21.01%	8.63%	6.13%	8.65%	12.22%	11.95%	13.48%	3.49%	1.03%
Cert. Transparency (1 key) [10]	12.7M	0.88%	3.75%	6.90%	1.49%	23.10%	8.69%	6.04%	7.99%	12.08%	11.78%	13.50%	3.04%	0.77%
PGP keyset (1 key) [30]	1.35M	0.44%	4.24%	4.09%	2.17%	13.91%	10.55%	7.18%	8.83%	14.34%	14.22%	16.79%	2.64%	0.59%

Table 3: *The ratio of resulting source groups identified by the classification method described in Section 4. Datasets are split into subsets based on the number of keys that can be attributed to a single source (batch). ‘-’ means no key was classified for the target group. ‘0.00%’ means that some keys were classified, but less than 0.005%.*

proximately 10.2%) was assigned to group X, which contains the Microsoft providers (CAPI, CNG, and .NET).

These estimates can be compared against the estimated distribution of commonly used web servers. Apache, Nginx, LiteSpeed, and Google servers with the OpenSSL library as the default option have a cumulative market share of 86% [32]. This value exhibits a remarkably close match to the classification rate obtained for OpenSSL (group V). MS Internet Information Services (IIS) is included with Microsoft’s cryptographic providers (group X) and has a market share of approximately 12%. Again, a close match is observed with the classification value of 10.2% obtained for users with 10-99 certificates certified within the same day (batch).

Users with 100 and more keys certified within the same day show an even stronger preference for OpenSSL library (89.9%; group V) and also for group VIII (4.6%; this group contains popular libraries such as OpenJDK’s SunRsaSign, Bouncy Castle and mbedTLS) at the expense of groups X and XI.

The classification accuracy for users with only single-key batches or a small number of keys per batch is significantly less certain, but the general trends observed for larger batches persist. Group V (OpenSSL) is most popular, with group X (Microsoft providers) being the second most common. Although we cannot obtain the exact proportions of keys generated using particular sources/groups, we can easily determine the proportion of keys that *certainly could not* have been generated by a given source by means of the occurrence of impossible values produced by the bit mask, i.e., values that are never produced by the given source. Using this method,

we can conclude for certain that 19%, 25%, 17% and 10% of keys for users with 1, 2-9, 10-99 and 100+ keys per batch, respectively, could not have been generated by the OpenSSL library (see [25] for details).

Another dataset of TLS keys was collected from Google’s Pilot Certificate Transparency server [10]. The dataset processing was the same as that for the previous TLS dataset [9]. For users with small numbers of keys (1 and 2-9), the general trends observed from the TLS IPv4 dataset were preserved. Interestingly, however, Certificate Transparency dataset indicates that group X (Microsoft) is significantly more popular (47%) than group V (OpenSSL) for users with 10-99 keys.

4.2.2 Sources of PGP keys

A different set of real-world keys can be obtained from PGP key servers [30]. We used a dump containing nearly 4.2 million keys, of which approximately 1.4 million were RSA keys suitable for classification using the same processing as for the TLS datasets. In contrast to the TLS handshakes, significantly fewer PGP keys could be attributed to the same batch (i.e., could be identified as originating from the same unknown source) based on the subject name and certification date. Still, 84 thousand unique keys were extracted in batches of 2-9 keys and 1 732 for batches of 10-99 keys.

The most prolific source group is group XI (which contains both libgcrypt from the GnuPG software distribution and the PGPSDK4 library), as seen in Table 3. This is intuitively expected because of the widespread use of these two software libraries. Group

VIII, consisting of the Bouncy Castle library (containing the *org.bouncycastle.openpgp* package), is also very common (36%) for batches of 10-99 keys.

Because of the lower accuracy of classification for users with smaller numbers of keys (1 and 2-9), it is feasible only to consider the general properties of these key batches and their comparison with the TLS case rather than being concerned with the exact percentage values in these settings. The results for the PGP dataset indicate a significant drop in the proportion of keys generated using the OpenSSL library. According to an analysis of the keys that *certainly could not* have been obtained from a given source, at least 47% of the single-key batches were certainly *not* generated by OpenSSL, and this percentage increases to 72% for batches of 2-9 keys. PGPSDK4 in FIPS mode (group IV) was found to be significantly more common than in the TLS datasets.

Note that an exported public PGP key usually contains a *Version* string that identifies the software used. Unfortunately, however, this might be not the software used to generate the original key pair but merely the software that was used to export the public key. If the public key was obtained via a PGP keyserver (as was the case for our dataset), then the *Version* string indicates the version of the keyserver software itself (e.g., *Version: SKS 1.1.5*) and cannot be used to identify the ratios of the different libraries used to generate the keys¹¹.

5 Practical impact of origin detection

The possibility of accurately identifying the originating library or card for an RSA key is not solely of theoretical or statistical interest. If some library or card is found to produce weak keys, then an attacker can quickly scan for other keys from the same vulnerable source. The possibility of detection is especially helpful when a successful attack against a weak key requires a large but practically achievable amount of computational resources. Preselecting potentially vulnerable keys saves an attacker from spending resources on all public keys.

The identification of the implementations responsible for the weak keys found in [3, 12] was a difficult problem. In such cases, origin classification can quickly provide one or a few of the most probable sources for further manual inspection. Additionally, a set of already identified weak keys can be used to construct a new classification group, which either will match an already known one (for which the underlying sources are known) or can be used to search for other keys that belong to this new group in the remainder of a larger dataset (even when the source is unknown).

¹¹A dataset with the original *Version* strings could be used to test these predictions.

Another practical impact is the decreased anonymity set of the users of a service that utilizes the RSA algorithm whose users are not intended to be distinguishable (such as the Tor network). Using different sources of generated keys will separate users into smaller anonymity groups, effectively decreasing their anonymity sets. The resulting anonymity sets will be especially small when individual users decides to use cryptographic hardware to generate and protect their private keys (if selected device does not fall into into group with widely used libraries). Note that most users of the Tor project use the default client, and hence the same implementation, for the generation of the keys they use. However, the preservation of indistinguishability should be considered in the development of future alternative clients.

Tor hidden services sometimes utilize ordinary HTTPS certificates for TLS [1], which can be then linked (via classification of their public keys) with other services of the same (unknown) operator.

Mixnets such as mixmaster and mixminion use RSA public keys to encrypt messages for target recipient and/or intermediate mix. If key ID is preserved, one may try to obtain corresponding public key from PGP keyserver and search for keys with the same source to narrow that user's anonymity set in addition to analysis like one already performed on alt.anonymous.messages [22]. Same as for Tor network, multiple seemingly independent mixes can be linked together if uncommon source is used to generate their's RSA keys.

A related use is in a forensic investigation in which a public key needs to be matched to a suspect key-generating application. Again, secure hardware will more strongly fingerprint its user because of its relative rarity.

An interesting use is to verify the claims of remote providers of Cryptography as a Service [4] regarding whether a particular secure hardware is used as claimed. As the secure hardware (cards) used in our analysis mostly exhibit distinct properties of their generated keys, the use of such hardware can be distinguished from the use of a common software library such as OpenSSL.

5.1 How to mitigate origin classification

The impact of successful classification can be mitigated on two fronts: by library maintainers and by library users. The root cause lies with the different design and implementation choices for key generation that influence the statistical distributions of the resulting public keys. A maintainer can modify the code of a library to eliminate differences with respect to the approach used by all other sources (or at least the most common one, which is OpenSSL in most cases). However, although this might

work for one specific library (mimicking OpenSSL), it is not likely to be effective on a wider scale. Changes to all major libraries by its maintainers are unlikely to occur, and many users will continue to use older versions of libraries for legacy reasons.

More pragmatic and immediate mitigation can be achieved by the users of these libraries. A user may repeatedly generate candidate key pairs from his or her library or device of choice and reject it if its classification is too successful. Expected number of trials differs based on the library used and the prior probability of sources within the targeted domain. For example, if TLS is the targeted domain, five or less key generation trials are expected for most libraries to produce “indecisive” key.

The weakness of the second approach lies in the unknown extent of public modulus leakage. Although we have described seven different causes of leakage, others might yet remain unknown – allowing for potential future classification of keys even after they have been optimized for maximal indecisiveness against these seven known causes.

This strategy can be extended when more keys are to be generated. All previously generated keys should be included in a trial classification together with the new candidate key. The selection process should also be randomized to some extent; otherwise, a new classification group of “suspiciously indecisive” keys might be formed.

6 Key generation process on cards

The algorithms used in open-source libraries can be inspected and directly correlated to the biases detected in their outputs. To similarly attribute the biased keys produced by cards to their unknown underlying algorithms, we first verified whether the random number generator might instead be responsible for the observed bias. We also examined the time- and power-consumption side channels of the cards to gain insight into the processes responsible for key generation.

Truly random data generated on-card are a crucial input for the primes used in RSA key pair generation. A bias in these data would influence the predictability of the primes. If a highly biased or malfunctioning generator is used, factorization is not necessary (only a small number of fixed values can be taken as primes) or is feasible even for RSA keys with lengths otherwise deemed to be secure [3, 6, 12].

6.1 Biased random number generator

The output of an on-card truly random number generator (TRNG) can be tested using statistical batteries, and deviances are occasionally detected in commercial security tokens [6]. We generated a 100 MB stream of ran-

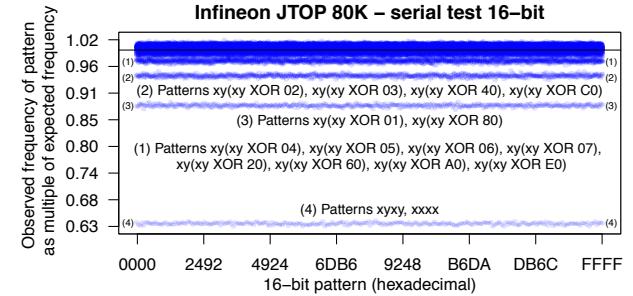


Figure 6: *The frequencies of different patterns with the length of 16 bits computed from 1 GB random data stream generated by the Infineon JTOP 80K card. At least five distinct patterns can be identified where all patterns should exhibit an uniform distribution instead.*

dom data from one card of each type and tested these data streams using the common default settings of the NIST STS and the Dieharder battery of statistical tests [7, 23] as well as our alternative EACirc distinguisher [24]. All types of cards except two (Infineon JTOP 80K and Oberthur Cosmo Dual 72K) passed the tests with the expected number of failures at a confidence level of 1%.

The Infineon JTOP 80K failed the NIST STS Approximate Entropy test (85/100, expected entropy contained in the data) at a significant level and also failed the group of Serial tests from the Dieharder suite (39/100, frequency of overlapping n -bit patterns). Interestingly, the serial tests began to fail only for patterns with lengths of 9 bits and longer (lengths of up to 16 bits were tested), suggesting a correlation between two consecutive random bytes generated by the TRNG. As shown in Figure 6, for 16-bit patterns, all bytes in the form of $xyxy$ (where x and y denote 4-bit values) were 37% less likely to occur than other combinations. At least three more distinct groups of inputs with smaller-than-average probabilities were also identified. Note that deviating distributions were observed in all three physical Infineon JTOP 80K cards that were tested and thus were probably caused by a systematic defect in the entire family of cards rather than a single malfunctioning device. The detected bias is probably not sufficient to enable faster factorization by guessing potential primes according to the slightly biased distribution. However, it may be used to identify this type of card as the source of a sufficiently large (e.g., 1KB) random data stream (i.e., to fingerprint such a random stream).

The Oberthur Cosmo Dual 72K failed more visibly, as two cards were blocked after the generation of only several MB of random data. The statistical tests then frequently failed because of the significant bias in the data. Several specific byte values were never produced in the “random” stream.

We also generated data streams directly from the concatenated exported primes with the two most significant bytes and the least two bits dropped, as the previous analysis had revealed a non-uniform distribution in these bits. Interestingly, both the Infineon JTOP 80K and the Oberthur Cosmo Dual 72K failed only for their random data streams (as described above) but successfully passed¹² for the streams generated from the concatenated primes, hinting at the possibility that either random data are generated differently during prime generation or (unlikely) the prime selection process is able to mask the bias observed in the raw random data.

6.1.1 Malfunctioning generator

All primes for the card-generated 512- and 1024-bit keys were tested for uniqueness. All tested card types except one generated unique primes. In the exceptional case of the Oberthur Cosmo Dual 72K cards, approximately 0.05% of the generated keys shared a specific value of prime q . The flaw was discovered in all three tested physical cards for both 512-bit and 1024-bit keys. The repeated prime value was equal to 0xC000...0077 for 512-bit RSA keys and 0xC000...00E9B for 1024-bit RSA keys. These prime values correspond to the first Blum prime generated when starting from the value 0xC000...000 in each case.

The probable cause of such an error is the following sequence of events during prime generation: 1) The random number generator of the card was called but failed to produce a random number, either by returning a value with all bits set to zero or by returning nothing into the output memory, which had previously been zeroed. 2) The candidate prime value q (equal to 0 at the time) had its highest four bits fixed to 1100₂ (to obtain a modulus of the required length¹³ when multiplied by the prime p), resulting in a value of 0xC0 in the most significant byte. 3) The candidate prime value was tested for primality and increased until the first prime with the required properties (a Blum prime in the case of the Oberthur Cosmo Dual 72K) was found (0xC000...0077 in the case of 512-bit RSA).

The faulty process described above that leads to the observed predictable primes may also occur for other cards or software libraries as a result of multiple causes (e.g., an ignored exception in random number generation or a programming error). We therefore inspected our key pair dataset, the TLS IPv4 dataset [9] and the PGP dataset [30] for the appearance of such primes relevant to key lengths of 512, 1024 and 2048 bits. Interestingly, no such corrupt keys were detected except for those already described.

¹²Except for the Oberthur nearly zero keys (see Section 6.1.1).

¹³As was observed for the dataset analysed in Section 3.

Note that a random search for a prime is much less likely to fail in this mode. Even if some of the top bits and the lowest bit are set to one, the resulting value is not a prime for common MSB masks. New values will be generated if the starting value contains only zeroes.

6.2 Power analysis of key generation

Analysis of power consumption traces is a frequently used technique for card inspection. The baseline power trace expected should cover at least the generation of random numbers of potential primes, primality testing, computation of the private exponent and storage of generated values into a persistent key pair object. We utilized the simple power analysis to reveal significant features like random number generation, RSA encryption, and RSA decryption operation, separately. By programming a card to call only the desired operation (generate random data, encrypt, decrypt), the feature pattern for the given operation is obtained. These basic operations were identified in all tested types of cards. Once identified, the operations can be searched for inside a more complex operations like the RSA key pair generation.

A typical trace of the RSA key pair generation process (although feature patterns may differ with card hardware) contains: 1) Power consumption increases after the generating key pair method is called (cryptographic RSA coprocessor turned on). 2) Candidate values for primes p and q are generated (usage of a TRNG can be observed from the power trace) and tested. 3) The modulus and the private exponent are generated (assumed, not distinguishable from the power trace). 4) Operation with a private key is executed (decryption, in 7 out of 16 types of cards) to verify key usability. 5) Operation with a public key is executed (encryption, 3 types of cards only).

Note that even when the key generation process is correctly guessed, it is not possible to simply implement it again and compare the resulting power traces – as only the card’s main CPU is available for user-defined operations, instead of a coprocessor used by the original process. Additional side-channel and fault induction protection techniques may be also applied. Therefore, one cannot obtain an exactly matching power trace from a given card due to unavailability of low-level programming interfaces and additionally executed operations for verification of key generation hypothesis.

Whereas some steps of the key generation, such as the randomness generation, take an equal time across multiple runs of the process, the time required to generate a prime differs greatly as can be also seen from the example given in Figure 7, where timing is extracted from the power trace. The variability can be attributed to the randomized process of the prime generation. Incremental search will find the first prime greater than a random

number selected as the base of the search. Since both primes p and q are distributed as distances from a random point to a prime number, the resulting time distribution will be affected by a mixture of these two distributions.

In samples collected from 12 out of 16 types of cards, the distribution of time is concentrated at evenly spaced points¹⁴ as seen in Figure 7. The distance between a pair of points is interpreted as the duration of a single primality test, whereas their amount corresponds to the number of candidates that were ruled out by the test as a composite. Then it is possible to obtain a histogram of number of tested candidates, e.g., by binning the distribution with breaks placed in the midpoints of the empty intervals.

6.3 Time distribution

We experimentally obtained distributions for a number of needed primality tests for different parameters of trial division. Then we were able to match them with distributions from several cards, obtaining a likely estimate for the number of primes used by the card in the trial division (sieving) phase. For some types of cards, a single parameter did not match distributions of neither 512-bit nor 1024-bit keys. There may exist a different optimal value of trial division tests and primality tests for different key lengths. Notably, in some cases of card-generated 512-bit keys, the number of primality tests would have to be halved to exactly match a referential distribution. However, we are not aware of a mechanism that would perform two primality tests in parallel or at least in the same time, as is required for testing a candidate of double bit length.

The exact time distribution for software implementations is of less concern since the key generation process tends to be much faster on an ordinary CPU. The source code can be modified to accommodate for counting the number of tests directly (as shown in the inlay in Figure 7) without relying on time measurement that may be influenced by other factors specific to the implementation.

7 Conclusions

This paper presents a thorough analysis of key pairs generated and extracted from 38 different sources encompassing open-source and proprietary software libraries and cryptographic cards. This broad analysis allowed us to assess current trends in RSA key pair generation even when the source codes for key generation were not available, as in the case of proprietary libraries and cards. The

¹⁴Due to small differences in duration of key generation and rounding caused by precision of the measurement, the times belonging to the same group will not be identical to one millisecond. The peaks were highlighted by summing adjoining milliseconds, but only in the case when large (almost) empty spaces exist in the distribution.

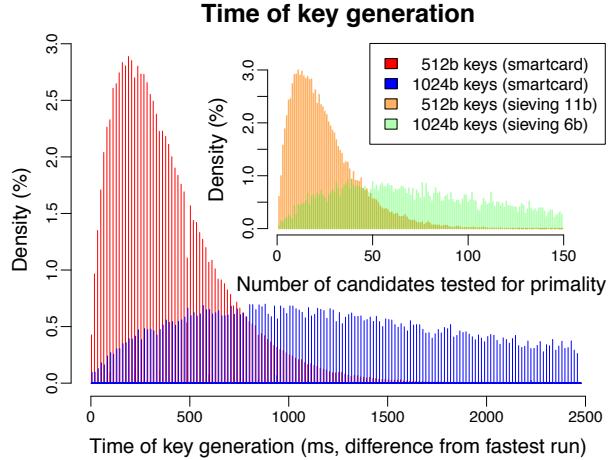


Figure 7: An example of the histogram of times necessary to generate a large number of 512 and 1024-bit RSA keys generated from an NXP J2D081 card. Left – the distribution of key generation times is concentrated around evenly spaced points, with the distance representing the duration of a single primality test. The times were normalized to begin at zero, therefore they represent difference from the fastest run. Inlay – the distribution of number of candidates tested by primality tests obtained from a software implementation. 512-bit keys are generated with trial division up to 11-bit primes, 1024-bit keys used 6-bit primes. The results show a clear correlation between the generation time and an expected number of primality tests.

range of approaches identified indicates that the question of how to generate an optimal RSA key has not yet been settled.

The tested keys were generally found to contain a high level of entropy, sufficient to protect against known factorization attacks. However, the source-specific prime selection algorithms, postprocessing techniques and enforcement of specific properties (e.g., Blum primes) make the resulting primes slightly biased, and these biases serve as fingerprints of the sources. Our paper therefore shows that public moduli leak significantly more information than previously assumed. We identified seven properties of the generated primes that are propagated into the public moduli of the generated keys. As a result, accurate identification of originating library or smartcard is possible based only on knowledge of the public keys. Such an unexpected property can be used to decrease the anonymity set of RSA keys users, to search for keys generated by vulnerable libraries, to assess claims regarding the utilization of secure hardware by remote parties, and for other practical uses. We classified the probable origins of keys in two large datasets consisting of 10 and

15 million (mostly) TLS RSA keys and 1.4 million PGP RSA keys to obtain an estimate of the sources used in real-world applications.

The random number generator is a crucial component for the generation of strong keys. We identified a generic failure scenario that produces weak keys and occasionally detected such keys in our dataset obtained from the tested cards. Luckily, no such weak key was identified in the datasets of publicly used RSA keys.

Acknowledgements: We acknowledge the support of the Czech Science Foundation, project GA16-08565S. The access to the computing and storage resources of National Grid Infrastructure MetaCentrum (LM2010005) is greatly appreciated. We would like to thank all anonymous reviewers and our colleagues for their helpful comments and fruitful discussions.

References

- [1] ARMA. Tor blog: Facebook, hidden services, and https certs. Available from <https://blog.torproject.org/blog/facebook-hidden-services-and-https-certs>, cit. [2016-06-26].
- [2] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., SIMIONATO, L., STEEL, G., AND TSAY, J.-K. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology – CRYPTO 2012: 32nd Annual Cryptology Conference. Proceedings*. Springer-Verlag, 2012, pp. 608–625.
- [3] BERNSTEIN, D. J., CHANG, Y.-A., CHENG, C.-M., CHOU, L.-P., HENINGER, N., LANGE, T., AND SOMEREN, N. Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild. In *Advances in Cryptology – ASIACRYPT 2013*. Springer-Verlag, 2013, pp. 341–360.
- [4] BERSON, T., DEAN, D., FRANKLIN, M., SMETTERS, D., AND SPREITZER, M. Cryptography as a network service. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)* (2001).
- [5] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98: 18th Annual International Cryptology Conference. Proceedings*. Springer-Verlag, 1998, pp. 1–12.
- [6] BOORGHANY, A., SARMADI, S., YOUSEFI, P., GORJI, P., AND JALILI, R. Random data and key generation evaluation of some commercial tokens and smart cards. In *Information Security and Cryptology (ISCISC), 11th International ISC Conference. Proceedings*. IEEE, 2014, pp. 49–54.
- [7] BROWN, R. G. Dieharder: A random number test suite, version 3.31.1, 2004. Available from: <http://www.phy.duke.edu/~rgb/General/dieharder.php>, cit. [2016-06-26].
- [8] BRUMLEY, B. B., AND TUVERI, N. Remote Timing Attacks Are Still Practical. In *Computer Security – ESORICS 2011: 16th European Symposium on Research in Computer Security. Proceedings*. Springer-Verlag, 2011, pp. 355–371.
- [9] DURUMERIC, Z., ET AL. Internet-Wide Scan Data Repository: Full IPv4 HTTPS Handshakes, dump from June 02, 2016. Available from: <https://scans.io/>, [cit. 2016-06-02].
- [10] GOOGLE. Certificate Transparency dump from June 07, 2016. <https://www.certificate-transparency.org>, cit. [2016-06-07].
- [11] GORDON, J. *Strong Primes are Easy to Find*. Springer-Verlag, 1985, pp. 216–223.
- [12] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *21st USENIX Security Symposium. Proceedings*. USENIX, 2012, pp. 205–220.
- [13] IEEE. Standard Specifications for Public-Key Cryptography. IEEE Std 1363, 2000.
- [14] KERRY, C. F., AND ROMINE, C. FIPS PUB 186-4 Digital Signature Standard (DSS), 2013.
- [15] KOCHER, P., JAFFE, J., AND JUN, B. Differential Power Analysis. In *Advances in Cryptology – CRYPTO'99: 19th Annual International Cryptology Conference. Proceedings*. Springer-Verlag, 1999, pp. 388–397.
- [16] LEHMAN, R. S. Factoring large integers. In *Mathematics of Computation*, vol. 28. American Mathematical Society, 1974, pp. 637–646.
- [17] LOEBENBERGER, D., AND NÜSKEN, M. Notions for RSA Integers. In *International Journal of Applied Cryptography*. InderScience Publishers, 2014, pp. 116–138.
- [18] MAURER, U. M. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology* 8, 3 (1995), 123–155.
- [19] MENEZES, A. J., OORSCHOT, P. C. V., VANSTONE, S. A., AND RIVEST, R. L. *Handbook of Applied Cryptography*, 1st ed. CRC Press, 1996.
- [20] MIRONOV, I. Factoring RSA Moduli II. Available from <https://windowsontopology.org/2012/05/17/factoring-rsa-moduli-part-ii/>, cit. [2016-06-26].
- [21] PULKUS, J. Efficient Prime-Number Check, Oct. 2 2014. US Patent App. 14/354,455.
- [22] RITTER, T. De-anonymizing alt.anonymous.messages. Available from <https://ritter.vg/p/AAM-defcon13.pdf>, cit. [2016-06-26].
- [23] RUKHIN, A., ET AL. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. In *NIST Special Publication 800-22rev1a*. NIST, 2010.
- [24] SÝS, M., ŠVENDA, P., UKROP, M., AND MATYÁŠ, V. Constructing empirical tests of randomness. In *SECRYPT 2014, SCITEPRESS* (2014), pp. 229–237.
- [25] ŠVENDA, P., NEMEC, M., SEKAN, P., KVAŠNOVSKÝ, R., FORMÁNEK, D., KOMÁREK, D., AND MATYÁŠ, V. *The Million-Key Question Investigating the Origins of RSA Public Keys*. Technical report FIMU-RS-2016-03. Masaryk University, Czech Republic, 2016.
- [26] WAGNER, D. Cryptanalysis of a Provably Secure CRT-RSA Algorithm. In *11th ACM Conference on Computer and Communications Security. Proceedings*. ACM, 2004, pp. 92–97.
- [27] WILLIAMS, H. C. A $p+1$ Method of Factoring. In *Mathematics of Computation*, vol. 39. American Mathematical Society, 1982, pp. 225–234.
- [28] ANSI X9.31-1998: Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry (rDSA), 1998.
- [29] YAFU: Yet Another Factorization Utility, 2013. Available from: <http://sourceforge.net/projects/yafu/>, cit. [2016-06-26].
- [30] PGP keydump from June 02, 2016. Available from: <http://pgp.key-server.io/dump/current/>, cit. [2016-06-02].
- [31] Keys collected in 1MRSA project, 2016. Available from: <http://crcs.cz/papers/usenix2016/1mrsaset>.
- [32] W3Techs Web Technology Surveys: Usage of web servers for websites, 2016. Available from http://w3techs.com/technologies/overview/web_server/all, cit. [2016-06-26].

Fingerprinting Electronic Control Units for Vehicle Intrusion Detection

Kyong-Tak Cho and Kang G. Shin

The University of Michigan

{*ktcho, kgshin*}@umich.edu

Abstract

As more software modules and external interfaces are getting added on vehicles, new attacks and vulnerabilities are emerging. Researchers have demonstrated how to compromise in-vehicle Electronic Control Units (ECUs) and control the vehicle maneuver. To counter these vulnerabilities, various types of defense mechanisms have been proposed, but they have not been able to meet the need of strong protection for safety-critical ECUs against in-vehicle network attacks. To mitigate this deficiency, we propose an anomaly-based intrusion detection system (IDS), called *Clock-based IDS* (CIDS). It measures and then exploits the intervals of periodic in-vehicle messages for fingerprinting ECUs. The thus-derived fingerprints are then used for constructing a baseline of ECUs' clock behaviors with the Recursive Least Squares (RLS) algorithm. Based on this baseline, CIDS uses Cumulative Sum (CUSUM) to detect any abnormal shifts in the identification errors — a clear sign of intrusion. This allows quick identification of in-vehicle network intrusions with a low false-positive rate of 0.055%. Unlike state-of-the-art IDSs, if an attack is detected, CIDS's fingerprinting of ECUs also facilitates a root-cause analysis; identifying which ECU mounted the attack. Our experiments on a CAN bus prototype and on real vehicles have shown CIDS to be able to detect a wide range of in-vehicle network attacks.

1 Introduction

Security has now become an important and real concern to connected and/or automated vehicles. The authors of [9] systematically analyzed different attack vectors in vehicles (e.g., Bluetooth, Cellular), and showed that in-vehicle Electronic Control Units (ECUs) can be compromised for remote attacks. Through a compromised ECU, the adversary can control the vehicle by injecting packets in the in-vehicle network [20, 23]. Researchers have

also been able to compromise and remotely stop a Jeep Cherokee running on a highway [7, 25], which triggered a recall of 1.4 million vehicles. Such a reality of vehicle attacks has made automotive security one of the most critical issues.

As a countermeasure against such attacks on in-vehicle networks, two main lines of defense have been pursued: *message authentication* and *intrusion detection*. Although message authentication provides a certain level of security and is shown to be efficient for Internet security, its adoption in in-vehicle networks is hindered by (i) the limited space available for appending a Message Authentication Code (MAC) in in-vehicle messages and (ii) its requirements of real-time processing and communication.

Various types of Intrusion Detection Systems (IDS) have been proposed [16, 23, 30, 31]. The essence of state-of-the-art IDSs is to monitor the contents and the periodicity of in-vehicle messages and verify whether there are any significant changes in them. Since they are either constant or predictable in in-vehicle networks, such approaches can be feasible in most circumstances. However, there still remain critical attacks which existing IDSs can neither detect nor prevent, for two main reasons: 1) in-vehicle messages do not carry information on their transmitters, and thus one cannot tell whether they originate from genuine transmitters; and 2) lack of the transmitters' information makes it very difficult or impossible for state-of-the-art IDSs to identify which ECU has mounted an attack.

To overcome these limitations and defend against various vehicle attacks, we propose a new anomaly-based IDS, called *Clock-based IDS* (CIDS). The need of CIDS for vehicles is motivated through an analysis of three representative in-vehicle network attacks — fabrication, suspension, and masquerade attacks. Our analysis shows that state-of-the-art IDSs are insufficient, especially in detecting the masquerade attack due to the absence of the transmitters' information in messages. CIDS over-

comes these limitations of existing IDSs by fingerprinting in-vehicle ECUs. Researchers have proposed various schemes for fingerprinting network devices by estimating their clock skews through the timestamps carried in their control packet headers [17, 19, 34, 42]. However, since such embedded timestamps are not available for in-vehicle networks making them inapplicable, CIDS fingerprints in-vehicle ECUs in a very different way.

CIDS monitors the intervals of (commonly seen) periodic in-vehicle messages, and then exploits them to estimate the clock skews of their transmitters which are then used to fingerprint the transmitters. That is, instead of assuming or requiring timestamps to be carried in messages for fingerprinting, CIDS exploits the periodic feature (seen at receivers) of in-vehicle network messages for fingerprinting transmitter ECUs. This makes CIDS invulnerable to attackers who use faked timestamps and thus clock skews — a problem that timestamp-based fingerprinting schemes cannot handle. Based on the thus-obtained fingerprints, CIDS constructs a norm model of ECUs’ clock behaviors using the Recursive Least Squares (RLS) algorithm and detects intrusions with a Cumulative Sum (CUSUM) analysis. This enables CIDS to detect not only attacks that have already been demonstrated or discussed in literature, but also those that are more acute and cannot be detected by state-of-the-art IDSs. Our experimental evaluations show that CIDS detects various types of in-vehicle network intrusions with a low false-positive rate of 0.055%. Unlike state-of-the-art IDSs, if an intrusion is detected in CIDS, its fingerprinting capability facilitates identification of the (compromised) ECU that mounted the attack. We validate these capabilities of CIDS through experimental evaluations on a CAN bus prototype and on real vehicles.

We focus on building CIDS for Control Area Network (CAN), which is the *de facto* standard in-vehicle network. Its applicability to other in-vehicle network protocols is also discussed in Section 6. Considering the ubiquity of CAN and its direct relationship with the drivers/passengers’ safety, it is critically important to build as capable a CAN bus IDS as possible.

This paper makes the following contributions:

- Development of a novel scheme of fingerprinting ECUs by exploiting message periodicity;
- Proposal of CIDS, which models the norm behavior of in-vehicle ECUs’ clocks based on fingerprints and then detects in-vehicle network intrusions;
- Implementation and validation of CIDS on a CAN bus prototype as well as on 3 real vehicles.

The rest of the paper is organized as follows. Section 2 provides the necessary background of CAN and IDS-related work, and Section 3 details the attack model



Figure 1: Format of a CAN data frame.

we consider. Section 4 details the design of CIDS, which is evaluated in Section 5 on a CAN bus prototype as well as on three real vehicles. Section 6 discusses CIDS further, such as its overhead and extension to emerging in-vehicle networks. Finally, we conclude the paper in Section 7.

2 Background

For completeness, we first provide necessary background on the CAN protocol, and then discuss the related work on security solutions for in-vehicle networks.

2.1 Primer on CAN Protocol

CAN frame. CAN is the most widely deployed in-vehicle communication protocol, which interconnects ECUs/nodes through a multi-master, message broadcast bus system [4]. To maintain data consistency and make control decisions, data is exchanged between ECUs via CAN frames, the format of which is shown in Fig. 1. A CAN frame contains fields such as ID, Data Length Code (DLC), Data, and CRC. Since CAN is message-oriented, instead of containing the transmitter/receiver address, a CAN frame contains a unique ID which represents its priority and meaning. For example, a frame with ID=0x20 may contain wheel speed values whereas a frame with ID=0x55 may contain temperature values.

Arbitration. Once the CAN bus is detected idle, nodes with buffered messages to transmit, attempt to access the bus. Multiple nodes could attempt to access the bus simultaneously, i.e., contention occurs for access. Such a contention is resolved via bus arbitration as follows. Each node first transmits the ID value of its CAN frame one bit at a time, starting with the most significant bit. Since CAN is designed to logically behave as a wired-AND gate, some contending nodes see an output of 0 from the bus, although they had transmitted 1. Such nodes withdraw from bus contention and switch to the receive mode. As a result, among the contending nodes, the ECU sending the message with the lowest ID value wins arbitration, and gains exclusive access for message transmission. Those which have lost arbitration re-attempt to transmit once the bus becomes idle again.

Synchronization. For proper bitwise message transmission and reception, hard and soft *bit* synchronizations are achieved, respectively, by using the Start-of-Frame (SOF) signal and bit stuffing in CAN frames [4]. Al-

though these provide alignment of bit edges for message exchange, they do not synchronize the *clocks* of ECUs, i.e., CAN lacks clock synchronization. Thus, since time instants for ECUs are provided by their own quartz crystal clocks, these clocks, in reality, run at different frequencies, resulting in random drifting of clocks: a drift of 2400ms over a period of 24 hours is possible [27].

2.2 Related Work

To defend against various types of vehicle cyber attacks, there have been two main streams of security solutions: *message authentication* and *intrusion detection*.

Message authentication. In the area of Internet security, cryptographic message authentication provides strong protection against forgery. Thus, researchers have attempted to borrow such approaches from the domain of Internet security to address in-vehicle network security problems. However, since the maximum payload length allowed in the CAN data field is only 8 bytes, the available space for appending a cryptographically secure Message Authentication Code (MAC) is very limited, i.e., the protocol specification limits its maximum cryptographic strength. To overcome this difficulty, rather than appending a MAC in one CAN frame’s data field, the authors of [38] proposed to truncate it across multiple frames. Instead of the data field, the authors of [33] proposed to use multiple CRC fields to include 64 bits of CBC-MAC. The authors of [15] suggested to exploit an out-of-band channel for message authentication.

Although such preventive measures provide some degree of security, they alone cannot guarantee complete security due to their inability to handle certain critical attacks, e.g., Denial-of-Service (DoS). Moreover, their operations not only require a significant amount of processing power but also increase message latencies and bus utilization. Since in-vehicle networks must operate in real time and ECUs are resource-limited for cost reasons, unlike in the Internet, these “costs” of preventive measures hinder their adoption [30]. More importantly, when an adversary has full access to any data stored in RAM and/or FLASH, including data used for implementing security mechanisms (e.g., shared secret keys), some cryptographic solutions become incapable [24].

Intrusion detection. To overcome such limitations of preventive measures, different Intrusion Detection Systems (IDSs) have been proposed. Some state-of-the-art IDSs exploit the fact that most CAN messages are periodic, i.e., sent at fixed time intervals. The authors of [30] proposed an IDS which monitors the intervals of periodic messages, measures their entropies, and exploits them for intrusion detection. Similarly, a method of modeling the distribution of message intervals, and utilizing it for intrusion detection was proposed in [23]. In addition

to message frequency, researchers also proposed to verify the message contents. The authors of [31] exploited in-vehicle sensors to verify message range, correlation, etc. Abnormal measurements on brake-related sensors were detected by using the tire-friction model [10].

Although existing IDSs are capable of detecting most attacks through the above approaches, they fail to cover some critical attacks which are more acute, and thus are not sufficient to provide security. We will elaborate on such shortcomings of state-of-the-art IDSs while analyzing the attack scenarios under consideration in Section 3.3.

3 Attack Model

We first discuss the adversary model under consideration, and then the three representative attack scenarios.

3.1 Adversary Model

Adversaries can physically/remotely compromise more than one in-vehicle ECU via numerous attack surfaces and means [9]. We consider an adversary who wants to manipulate or impair in-vehicle functions. The adversary can achieve this by either injecting arbitrary messages with a spoofed ID into the in-vehicle network, which we refer to as *attack messages*, or by stopping/suspending message transmissions of the compromised ECU.

Strong and weak attackers. Depending on their hardware, software, and attack surfaces, ECUs of different vehicles have different degrees of vulnerabilities, thus providing attackers different capabilities. So, we consider two different types of compromised ECUs: *fully* and *weakly* compromised ECUs.

Through a *weakly* compromised ECU, the attacker is assumed to be able to stop/suspend the ECU from transmitting certain messages or keep the ECU in listen-only mode, but cannot inject any fabricated messages. We call such an attacker with limited capabilities a *weak attacker*, and will use this term interchangeably with “weakly compromised ECU”.

In contrast, with a *fully* compromised ECU, the attacker is assumed to have full control of it and access to memory data. Thus, in addition to what a weak attacker can do, the attacker controlling a fully compromised ECU can mount attacks by injecting arbitrary attack messages. We call such an attacker with more attack capabilities a *strong attacker*, and will use this term interchangeably with a “fully compromised ECU”. Even when preventive security mechanisms (e.g., MAC) are built into the ECUs, since the strong attacker has full access to any data stored in their memory, including data used for implementing security mechanisms (e.g., shared secret keys), it can disable them [24]. On the other hand,

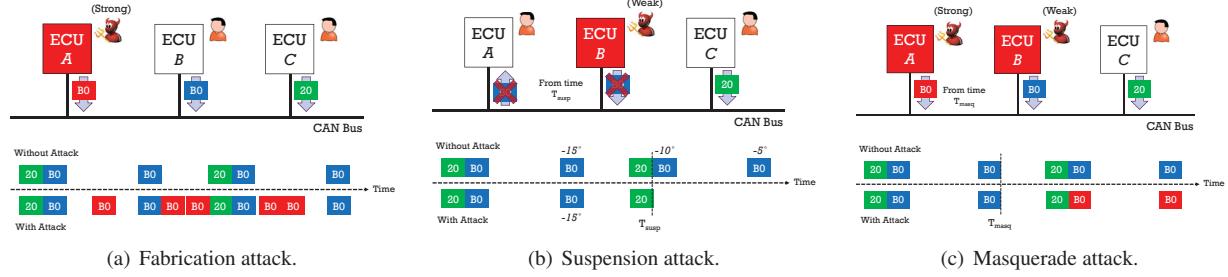


Figure 2: Three representative attack scenarios on in-vehicle networks.

a weak attacker can only stop, or listen to message transmissions, but cannot start a new one.

Foster *et al.* [13] have recently proved the possible existence of these two types of attackers in in-vehicle networks. They have shown that the firmware versions of telematics units can affect/limit the attacker's capabilities in injecting and monitoring in-vehicle network messages. Specifically, for a certain firmware version of the telematics unit, an attacker having control of that ECU was shown to be able to receive CAN messages but unable to send the messages. On the other hand, for some other firmware versions, the attacker was capable of both sending and receiving CAN messages to and from the in-vehicle network. In other words, the firmware version of an ECU determines which type of an attacker — strong or weak — it can become, if compromised.

To further comprehend how and why these two different types of attackers can exist, let's consider one of the most common CAN controllers, Microchip MCP2515 [1]. For ECUs with such a controller, various operation modes like configuration, normal, and listen-only can be selected by user instructions through the Serial Peripheral Interface (SPI). Thus, user-level features for configuring the CAN controller allow attackers to easily enter different modes (e.g., listen-only mode for a weak attacker). In contrast, there are no such features allowing attackers to easily inject forged messages. In other words, the specification of the ECU hardware/software, if compromised, can *restrict* the adversary to become a weak attacker only. Note that the required functionalities of a strong attacker subsume those of a weak attacker. It is thus easier for an adversary to become a weak attacker than a strong attacker, let alone researchers have already demonstrated how to create such a strong attacker [9, 20, 23, 24].

3.2 Attack Scenarios

Based on the adversary model discussed so far, we consider the following attack scenarios that can severely impair in-vehicle functions: *fabrication*, *suspension*, and

*masquerade.*¹

Fabrication attack. Through an in-vehicle ECU compromised to be a strong attacker, the adversary fabricates and injects messages with forged ID, DLC, and data. The objective of this attack is to override any periodic messages sent by a legitimate safety-critical ECU so that their receiver ECUs get distracted or become inoperable. For example, as shown in Fig. 2(a), the strong attacker \mathbb{A} injects several attack messages with ID=0xB0, which is usually sent by a legitimate ECU \mathbb{B} , at a high frequency. Thus, other nodes which normally receive message 0xB0 are forced to receive the fabricated attack messages more often than the legitimate ones. We refer to such a case as \mathbb{A} mounting a fabrication attack on message 0xB0 or its genuine transmitter \mathbb{B} . Demonstrated attacks such as controlling vehicle maneuver [20] and monopolizing the CAN bus with highest priority messages [16] exemplify a fabrication attack.

Suspension attack. To mount a suspension attack, the adversary needs only one weakly compromised ECU, i.e., become a weak attacker. As one type of Denial-of-Service (DoS) attack, the objective of this attack is to stop/suspend the weakly compromised ECU’s message transmissions, thus preventing the delivery/propagation of information it acquired, to other ECUs. For some ECUs, they must receive certain information from other ECUs to function properly. Therefore, the suspension attack can harm not only the (weakly) compromised ECU itself but also other receiver ECUs. An example of this attack is shown in Fig. 2(b) where the weak attacker having control of the Electric Power Steering ECU \mathbb{B} stops transmitting its measured steering wheel angle value. So, the Electronic Stability Control (ESC) ECU \mathbb{A} , which requires the steering wheel angle value from \mathbb{B} for detecting and reducing the loss of traction, no longer receives its updates and thus malfunctions.

Masquerade attack. To mount a masquerade attack, the adversary needs to compromise two ECUs, one as a strong attacker and the other as a weak attacker. The

¹In this paper, we focus on only these three attack scenarios and do not consider others as they may be less feasible or harmful, or be detectable by existing IDSs.

objective of this attack is to manipulate an ECU, while shielding the fact that an ECU is compromised. Fig. 2(c) shows an example where the adversary controls a strong attacker \mathbb{A} and a weak attacker \mathbb{B} . Until time T_{masq} , the adversary monitors and learns which messages are sent at what frequency by its weaker attacker, e.g., \mathbb{B} sends message 0xB0 every 20ms. Since most in-vehicle network messages are periodic and broadcast over CAN, it is easy to learn their IDs and intervals. Once it has learned the ID and frequency of a message, at time T_{masq} , the adversary stops the transmission of its weak attacker and utilizes its strong attacker \mathbb{A} to fabricate and inject attack messages with ID=0xB0. Stopping \mathbb{B} 's transmission and exploiting \mathbb{A} for transmission of attack messages are to overcome the weak attacker's inability of injecting messages. After T_{masq} , the original transmitter of 0xB0, \mathbb{B} , does not send that message any longer, whereas \mathbb{A} sends it *instead* at its original frequency. So, when the CAN bus traffic is observed, the frequency of message 0xB0 remains the same, whereas its transmitter has changed. We refer to such a case as \mathbb{A} mounting a masquerade attack on message 0xB0 or its original transmitter \mathbb{B} .

In fact, in order to attack and remotely stop a Jeep Cherokee running on a highway, Miller *et al.* [25] had to control its ABS collision prevention system by mounting a *masquerade* (not fabrication) attack. In contrast to other vehicles which they had previously examined (e.g., Toyota Prius), the Jeep Cherokee's brake was not controllable via the fabrication attack as its ABS collision prevention system, which was the attack vector for engaging brakes, was switched off when the fabrication attack was mounted. On the other hand, when mounting the masquerade attack, the system was not switched off, thus allowing them to control the Jeep Cherokee's braking maneuver.

Using masquerade attacks, the adversary can not only inject attack messages from the compromised/impersonating ECU but also cause other severe problems, significantly degrading the in-vehicle network performance. The impersonating ECU sending a message instead of another ECU implies that it would generate more messages to periodically transmit than before, making its transmit buffer more likely overloaded. This may, in turn, lead to severe consequences, such as non-abortable transmission requests [12], deadline violation [18], and significant priority inversion [32]. Moreover, the original sequence of messages may also change, thus failing to meet the requirement of some in-vehicle messages to be sent sequentially in a correct order for proper vehicle operations. These network problems from a masquerade attack occur while not deviating much from the norm network behavior (e.g., message frequency remains the same). This is in sharp contrast to the cases of mounting a fabrication attack

or a suspension attack, which may also incur similar problems. Such problems have been identified to be critical since they degrade the real-time performance of CAN significantly, and thus undermine vehicle safety [12, 18, 32]. The masquerade attack can thus cause more problems to the in-vehicle network than just injecting attack messages.

3.3 Defense Against the Attacks

When the fabrication or suspension attack is mounted, the frequency of certain messages significantly and abnormally increases or decreases, respectively. Thus, if state-of-the-art IDSs [16, 23, 30, 31], which monitor the message frequencies, were to be used, the attacks can be detected.

When mounting the masquerade attack, however, the adversary does not change the original frequency of messages. Thus, the adversary may use this attack to evade state-of-the-art IDSs. Moreover, if the adversary does not change the content of messages as well, it can behave like a legitimate ECU. However, the adversary may later mount other types of attacks (e.g., a fabrication attack) through the impersonating ECU. Hence, defending against the masquerade attack implies not only detecting the attack reactively, but also preventing other attacks *proactively*.

4 Clock-Based Detection

Although state-of-the-art IDSs are capable of detecting some basic attacks such as fabrication attack and suspension attack, they fail to detect more sophisticated ones such as the masquerade attack for the following reasons.

- *No authenticity* — CAN messages lack information on their transmitters. So, existing IDSs do not know whether or not the messages on the CAN bus were sent by the genuine transmitter, and hence cannot detect any changes of the message transmitter.
- *Inability of identifying a compromised ECU* — Lack of the transmitter's information makes it very difficult or impossible for state-of-the-art IDSs to identify which of compromised ECUs mounted an attack.

If CAN frames do not carry any information on their transmitters, how could an IDS identify them and detect intrusions such as the masquerade attacks? Which behavior of CAN should the IDS model for detection of such intrusions? We answer these questions by developing a novel IDS, CIDS, which exploits message frequency to fingerprint the transmitter ECUs, and models a norm behavior based on their fingerprints for intrusion

detection. We focus on detecting intrusions in *periodic* messages as most in-vehicle messages are sent periodically [32, 36].

4.1 Fingerprinting ECUs

For each in-vehicle ECU in CAN, the time instants of periodic message transmissions are determined by its quartz crystal clock [27]. We follow the nomenclature of clocks of the NTP specification [26] and Paxson [35]. Let C_{true} be a “true” clock which reports the true time at any moment and C_i be some other non-true clock. We define the terms “clock offset, frequency, and skew” as follows.

- **offset:** difference in the time reported by clock C_i and the true clock C_{true} . We define *relative offset* as the offset between two non-true clocks.
- **frequency:** the rate at which clock C_i advances. Thus, the frequency at time t is $C'_i(t) \equiv dC_i(t)/dt$.
- **skew:** difference between the frequencies of clock C_i and the true clock C_{true} . We define *relative skew* as the difference in skews of two non-true clocks.

If two clocks have relative offset and skew of 0, then they are said to be *synchronized*. Otherwise, we consider they are *unsynchronized*. Since CAN lacks clock synchronization, it is considered to be unsynchronized.

Clock skew as a fingerprint. The clock offsets and skews of unsynchronized nodes depend solely on their local clocks, thus being distinct from others. As others have also concluded [17, 19, 42], clock skews and offsets can therefore be considered as fingerprints of nodes. Various studies have exploited this fact to fingerprint physical devices [17, 19, 34, 42]. However, they are not applicable to our problem as they exclusively rely on the timestamps carried in the packet headers, which are, as discussed before, not available in in-vehicle networks. Kohno *et al.* [19] considered an alternative to embedded timestamps: using Fourier Transform for clock skew estimation. However, as their approach relies on the unique characteristics of the Internet (e.g., multi-hop delays, large network topology), it cannot be directly applied to in-vehicle networks.

To build an effective IDS, which can detect various types of attack including the masquerade attack, it should be capable of verifying the transmitter of each message. However, since such information is not present in CAN messages, we must fingerprint ECUs with other “leaked” information. Unlike the existing approaches that exploit embedded timestamps, we exploit *message periodicity* to extract and estimate the transmitters’ clock skews, which are then used to fingerprint the transmitter ECUs.

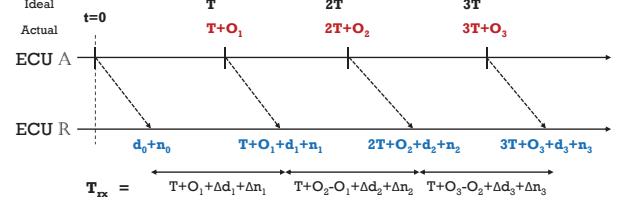


Figure 3: Timing analysis of message arrivals.

Clock skew estimation. Consider an ECU \mathbb{A} which broadcasts a message every T ms and an ECU \mathbb{R} which periodically receives that message. From the perspective of \mathbb{R} , since only its timestamp is available, we consider its clock as the true clock. As shown in Fig. 3, due to the clock skew, periodic messages are sent at times with small offsets from the ideal values (e.g., T , $2T$, $3T$, \dots). Let $t = 0$ be the time when the first message was sent from \mathbb{A} , and O_i be the clock offset of \mathbb{A} when it sends the i -th message since $t = 0$. Then, after a network delay of d_i , ECU \mathbb{R} would receive that message and put an arrival timestamp of $iT + O_i + d_i + n_i$, where n_i denotes the noise in \mathbb{R} ’s timestamp quantization [42]. Thus, the intervals between each arrival timestamp, $T_{rx,i} = T + \Delta O_i + \Delta d_i + \Delta n_i$, where ΔX_i denotes the difference of X between step i and $i - 1$, and $O_0 = 0$. Since the change in O_i within one time step is negligible and n_i is a zero-mean Gaussian noise term [2], the expected value of the timestamp intervals, $\mu_{T_{rx}} = E[T_{rx,i}]$, can be expressed as:

$$\begin{aligned} \mu_{T_{rx}} &= E[T + \Delta O_i + \Delta d_i + \Delta n_i] \\ &= T + E[\Delta O_i + \Delta d_i + \Delta n_i] \\ &\approx T, \end{aligned} \quad (1)$$

where the second equality holds since T is a pre-determined constant. Since the data lengths of CAN periodic messages, i.e., DLCs, are constant over time, for now, we consider $E[\Delta d_i] = 0$. Later in Section 4.4, we will discuss the case when d_i is not constant, and how it may affect the performance of CIDS.

Based on the arrival timestamp of the first message, $d_0 + n_0$, and the average of timestamp intervals, $\mu_{T_{rx}}$, we extrapolate and determine the *estimated* arrival time of the i -th message as $i\mu_{T_{rx}} + d_0 + n_0$, whereas the actual *measured* arrival time is $iT + O_i + d_i + n_i$. As we are estimating subsequent arrival times, $\mu_{T_{rx}}$ is determined by past measurements. Since T is constant over time and thus again $\mu_{T_{rx}} \approx T$, the average difference between the estimated and measured times is:

$$E[\mathbb{D}] = E[i(T - \mu_{T_{rx}}) + O_i + \Delta d + \Delta n] \approx E[O_i]. \quad (2)$$

That is, from message periodicity, we can estimate the *average clock offset*, $E[O_i]$, which will indeed be distinct for different transmitters. Since clock offset is

Algorithm 1 Clock skew estimation with RLS

```

1: Initialize:  $S[0] = 0$ ,  $P[0] = \delta I$ 
2: function SKEWUPDATE( $t, e$ ) ▷ RLS algorithm
3:    $G[k] \leftarrow \frac{\lambda^{-1}P[k-1]t[k]}{1+\lambda^{-1}t^2[k]P[k-1]}$ 
4:    $P[k] \leftarrow \lambda^{-1}(P[k-1] - G[k]t[k]P[k-1])$ 
5:   return  $S[k] \leftarrow S[k-1] + G[k]e[k]$ 
6: end function
7: for  $k^{th}$  step do
8:    $a_0 \leftarrow$  arrival timestamp of most recently rxed message
9:    $n \leftarrow 1$ 
10:  while  $n \leq N$  do
11:    if current time  $\gg a_{n-1}$  then
12:      /* No longer receives the message */
13:       $a_n, \dots, a_N \leftarrow$  significantly high values
14:       $T_n, \dots, T_N \leftarrow$  significantly high values
15:      break
16:    else
17:       $a_n \leftarrow$  arrival timestamp of  $n^{th}$  message
18:       $T_n \leftarrow a_n - a_{n-1}$  ▷ Timestamp interval
19:       $n \leftarrow n + 1$ 
20:    end if
21:  end while
22:   $\mu_T[k] \leftarrow \frac{1}{N} \sum_{i=1}^N T_i$  ▷ Avg. timestamp interval
23:   $O[k] \leftarrow \frac{1}{N-1} \sum_{i=2}^N a_i - (a_1 + (i-1)\mu_T[k-1])$ 
24:   $O_{acc}[k] \leftarrow O_{acc}[k-1] + |O[k]|$  ▷ Accumulated offset
25:   $e[k] \leftarrow O_{acc}[k] - S[k-1]t[k]$  ▷ Identification error
26:   $S[k] \leftarrow$  SKEWUPDATE( $t, e$ ) ▷ Clock skew
27: end for

```

slowly varying and non-zero [17, 42], $E[O_i] \neq 0$, whereas $E[\Delta O_i] = 0$.

If ECU \mathbb{R} were to determine the average clock offset for every N received messages, since it is derived in reference to the first message (of N messages), it represents only the average of *newly* incurred offsets. Thus, to obtain the total amount of incurred offset, which we call the *accumulated clock offset*, the absolute values of the average clock offsets have to be summed up. By definition, the slope of the accumulated clock offset would thus represent the clock skew, which is constant as we will show and as others have also concluded [19, 29, 35, 40]. This enables CIDS to estimate the clock skew from arrival timestamps and thus fingerprint the message transmitter for intrusion detection. We will later show, via experimental evaluations on a CAN bus prototype and on 3 real vehicles, that the thus-derived clock skew is indeed a fingerprint of an in-vehicle ECU.

4.2 CIDS — Per-message Detection

By determining the clock skew from observation of message intervals, transmitter ECUs can be fingerprinted. We exploit this in designing CIDS, a clock-based IDS for in-vehicle networks which detects intrusions in two different ways: *per-message* detection and *message-*

pairwise detection, where the latter supplements the former in reducing false positive/negative results. Next, we first discuss per-message detection and then pairwise detection.

Modeling. For a given message ID, CIDS derives the accumulated clock offset inherent in the arrival timestamps. Since clock skew is constant, the accumulated clock offset is linear in time, and hence CIDS describes it as a linear regression model. A linear parameter identification problem is thus formulated as:

$$O_{acc}[k] = S[k] \cdot t[k] + e[k], \quad (3)$$

where at step k , $O_{acc}[k]$ is the accumulated clock offset, $S[k]$ the regression parameter, $t[k]$ the elapsed time, and $e[k]$ the identification error. The regression parameter S represents the slope of the linear model and thus the *estimated clock skew*. The identification error, e , represents the residual which is not explained by the model. In CIDS, O_{acc} , S , t , and e are updated every N messages, i.e., KN messages are examined up to step k .

To determine the unknown parameter S , we use the Recursive Least Squares (RLS) algorithm [14], which uses the residual as an objective function to minimize the sum of squares of the modeling errors. Hence, in RLS, the identification error skews towards 0, i.e., has 0 mean. We will discuss the computational overhead of RLS as well as other possible solutions in Section 6.

Algorithm 1 describes how the clock skew is estimated using RLS. First, CIDS measures the arrival times and their intervals of N messages for a given ID. If the intended message has not been received for a long time — possibly due to suspension attack — as in line 13–14, CIDS sets the remaining timestamp and interval values significantly higher. Once N values are measured, CIDS determines the accumulated clock offset and accordingly, the identification error. Based on the thus-derived value, the gain, G , and the covariance, P , are updated with RLS for identifying the regression parameter S , i.e., estimate clock skew. This procedure of clock skew estimation continues iteratively during the operation of CIDS and, if uncompromised, outputs an identification error skewed towards 0 and a constant clock skew. This way, the *norm clock behavior* of the transmitter can be described as a linear model with the clock skew being the slope. In RLS, a forgetting factor, λ , is used to give exponentially less weights to older samples and thus provide freshness. In CIDS, we set $\lambda=0.9995$.

Detection. For a given message ID, CIDS runs RLS for clock skew estimation, constructs a norm model on clock behavior, and verifies whether there are any abnormal measurements deviating from it, i.e., intrusions.

Consider a fabrication attack in which the adversary injects an attack message with ID=0x01, which is originally sent every 10ms by some ECU. The fabrication

attack significantly increases the absolute average difference between the estimated and measured arrival times of 0x01. As a result, due to a sudden increase in the rate at which the accumulated clock offset changes, a high identification error results. Similarly, when the suspension attack is mounted, the absolute average difference also increases and thus a high error is also incurred. When a masquerade attack is mounted, since the adversary sends the message through a different ECU than its original one, the increase rate of accumulated clock offset, i.e., clock skew, suddenly changes and also results in a high identification error. In summary, unlike when the mean of identification error should usually skew towards 0, which is the norm clock behavior, its mean suddenly shifts towards a high non-zero value when there is an intrusion.

CIDS exploits the Cumulative Sum (CUSUM) method, which derives the cumulative sums of the deviations from a target value to detect sudden shifts. Since it is cumulative, even minor drifting from the target value leads to steadily increasing or decreasing cumulative values. It is therefore optimal in detecting small persistent changes and is widely used for change-point detection [8]. CIDS detects intrusions via CUSUM as follows. At each step of clock skew estimation, CIDS updates the mean and variance of the identification errors (e), μ_e and σ_e^2 , respectively. In CIDS, these values represent the CUSUM target values of e (i.e., norm clock behavior), and thus require proper tracking. Hence, as a precaution of abnormal values incurring from an attack to be reflected into the target values, μ_e and σ_e^2 are updated only if $|\frac{e-\mu_e}{\sigma_e}| < 3$. Then, per derived identification error e , the upper and lower control limits of CUSUM, L^+ and L^- are updated as [41]:

$$\begin{aligned} L^+ &\leftarrow \max [0, L^+ + (e - \mu_e)/\sigma_e - \kappa] \\ L^- &\leftarrow \max [0, L^- - (e - \mu_e)/\sigma_e - \kappa] \end{aligned} \quad (4)$$

where κ is a parameter reflecting the number of standard deviations CIDS intends to detect. Note that κ can be learned offline, or by monitoring normal in-vehicle traffic. If either of the control limits, L^+ or L^- , exceeds a threshold Γ_L , a sudden positive or negative shift in value has been detected, respectively, and thus CIDS declares it as an intrusion. As the general rule of thumb for CUSUM is to have a threshold of 4 or 5 [28], we set $\Gamma_L = 5$.

4.3 CIDS — Message-pairwise Detection

In addition to per-message detection, CIDS also alarms intrusions via message-pairwise detection, which examines the *correlation* between the average clock offsets in two periodic messages. Consider two messages M_1 and M_2 periodically sent by an ECU A. Since these messages

originate from the same transmitter, their instantaneous average clock offsets are likely equivalent. Thus, the correlation coefficient, ρ , between their average clock offsets (derived per step) would show a high value close to 1, i.e., correlated. On the other hand, if the two messages were sent by different ECUs, $\rho \approx 0$, i.e., uncorrelated.

Modeling and detection. If clock offsets in two messages are highly correlated ($\rho > 0.8$), their relationship can be linear. So, CIDS describes them as a linear regression model: $O_{M_2}[k] = \alpha O_{M_1}[k] + e_{corr}[k]$, where O_{M_i} denotes the average clock offset of message M_i at step k , α the regression parameter, and $e_{corr}[k]$ the identification error. As per-message detection, message-pairwise detection is also based on a linear model. Thus, we apply the same detection method, CUSUM. Since message-pairwise detection seeks intrusions from a different perspective than per-message detection, it reduces false positive/negative results. Note, however, that message-pairwise detection is only applicable when two messages' clock offsets are highly correlated, whereas per-message detection is applicable to any periodic message. Moreover, albeit effective, it requires pairwise computations. Therefore, we use message-pairwise detection as an *optional* feature of CIDS. We will later show via experimental evaluations how message-pairwise detection further improves the performance of CIDS.

4.4 Verification

To reduce possible false positives/negatives, CIDS also performs a verification process. Suppose that a possible intrusion was alarmed due to a high identification error when verifying message V_i , the i -th message of V . Although such a high error can be due to an intrusion, it can also be due to an incorrect computation of average clock offset. In Section 4.1, we considered $E[\Delta d_i] = 0$ and could thus extract and determine the average clock offset. Although this is true in most cases, occasionally $E[\Delta d_i] \neq 0$, which affects the accuracy of deriving the true clock offset and thus the detection result. In CAN, $E[\Delta d_i] \neq 0$ only occurs if the transmission of V_i was delayed due to the bus being busy or its transmitter losing arbitration when attempting to send V_i . Note that the latter also results in the bus being busy before the transmission/reception of V_i . Thus, CIDS also checks if the possibility of $E[\Delta d_i] \neq 0$ is the main cause of a (possibly false) alarm of intrusion by verifying whether the CAN bus was busy right before receiving V_i . This way, CIDS enhances its detection accuracy. However, as discussed before, usually $E[\Delta d_i] = 0$ in an actual CAN bus due to its high speed, its messages having short lengths, and low bus load. In other words, the nature of CAN bus communication helps CIDS reduce false positives/negatives.

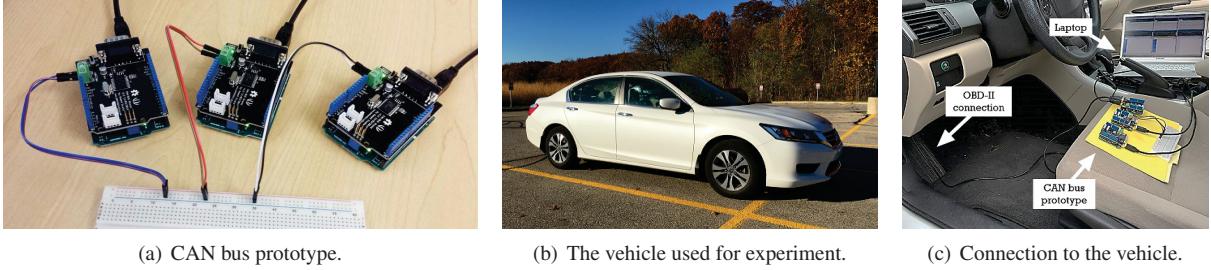


Figure 4: Different evaluation settings: (a) CAN bus prototype; (b) Honda Accord 2013 used for experiments on real vehicle; and (c) three prototype nodes communicating with real ECUs through the OBD-II port.

4.5 Root-cause Analysis

When an intrusion is detected for some message ID, CIDS can also identify which compromised ECU mounted the attack. It can extract the clock skew for that attacked message ID, compare it with other clock skew values extracted from other message IDs, and exploit the comparison result in determining whether they originated from the same transmitter. This way, CIDS can at least reduce the scope of ECUs which may (or may not) have mounted the attack, thus facilitating a root-cause analysis.

5 Evaluation

We now validate that clock skews can be used as fingerprints of transmitter ECUs, and evaluate the performance of CIDS on a CAN bus prototype and real vehicles.

CAN bus prototype: As shown in Fig. 4(a), we built a prototype with 3 CAN nodes, each of which consists of an Arduino UNO board and a SeeedStudio CAN shield. The CAN bus shield consists of a Microchip MCP2515 CAN controller, MCP2551 CAN transceiver, and a 120Ω terminal resistor to provide CAN bus communication capabilities. This prototype was set up to operate at a 500Kbps bus speed as in typical CAN buses. The first node \mathbb{A} was programmed to send messages 0x11 and 0x13 every 50ms, and the second node \mathbb{B} to send message 0x55 at the same frequency. The third node \mathbb{R} was programmed to run CIDS.

Real vehicle: A 2013 Honda Accord (Fig. 4(b)) is also used for our experiments in an isolated and controlled (for safety) environment. As shown in Fig. 4(c), via the On-Board Diagnostic (OBD-II) system port [3], we connected our CAN bus prototype nodes — which function as an adversary or CIDS — to the in-vehicle network. Through the OBD-II port, the three nodes were able to communicate with real ECUs.

CAN log data: To further validate that CIDS’s fingerprinting is applicable to other vehicles, we also refer to CAN traffic data logged from a Toyota Camry

2010 by Ruth *et al.* [36] and data logged from a Dodge Ram Pickup 2010 by Daily [11]. Both data were logged through a Gryphon S3 and Hercules software. In the Toyota Camry 2010, there were 42 distinct messages transmitted on the CAN bus: 39 of them sent periodically at intervals ranging from 10ms to 5 seconds, and 3 of them sent sporadically. In the Dodge Ram Pickup 2010, there were 55 distinct messages which were all sent periodically on the CAN bus.

In order to identify which messages originate from the same real ECU and thus exploit it as a ground truth, we used the naive method discussed in [32]. The messages, which originate from the same ECU and have the same preset message interval, were shown to have the same number of transmissions on the bus, when traced for at least a few minutes. Such a method can be an alternative to fingerprinting, but it requires pairwise comparisons and cannot be completed in real time as required in the design of CIDS, which is essential for intrusion detection in real in-vehicle networks.

While running CIDS, we determined offsets and skews for every 20 received samples, i.e., $N = 20$, and set $\kappa = 5$.

5.1 Clock Skew as a Fingerprint

We first evaluate the validity of CIDS’s fingerprinting of the transmitter ECUs based on the estimated clock skews. We evaluate skew estimates in microseconds per second ($\mu s/s$) or parts per million (ppm).

CAN bus prototype. Fig. 5(a) plots our evaluation results of CIDS’s fingerprinting on the CAN bus prototype: accumulated clock offsets of messages 0x11, 0x13, and 0x55. Note that the slopes in this figure represent the estimated clock skews. All the derived accumulated clock offsets were found to be linear in time, i.e., constant estimated skews. Messages 0x11 and 0x13, both of which were sent from node \mathbb{A} , exhibited the same constant clock skew of 13.4ppm. On the other hand, the message 0x55 sent from a different node \mathbb{B} showed a different clock skew of 27.2ppm. Thus, the clock skews derived by CIDS can be used to differentiate ECUs.

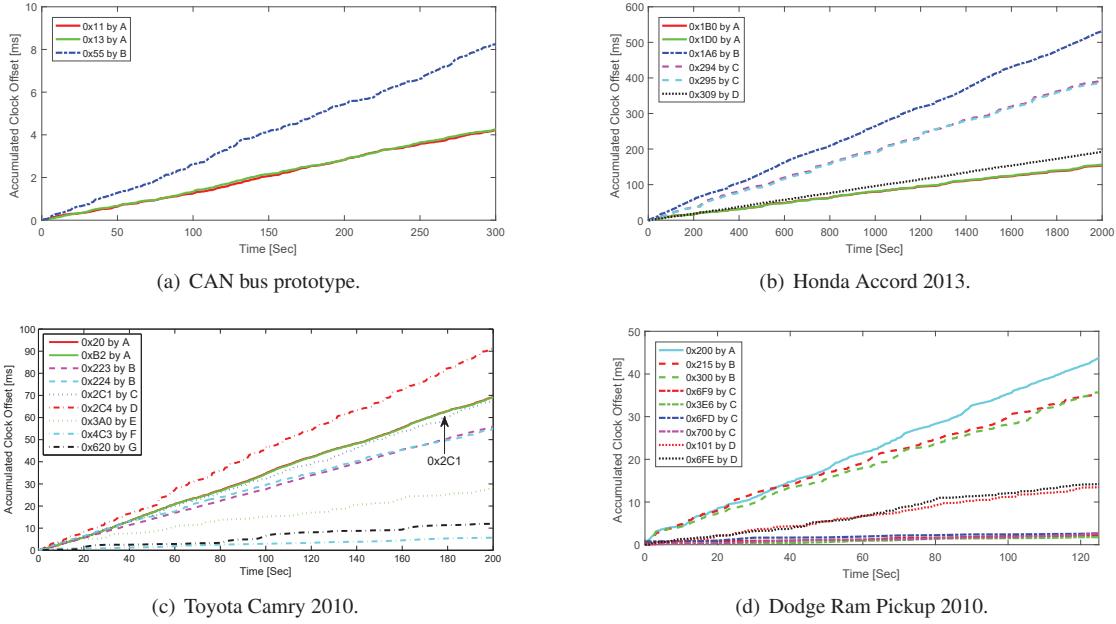


Figure 5: Accumulated clock offsets derived by CIDS in different evaluation settings.

Honda Accord 2013. For CIDS’s evaluation on a real vehicle, the CAN prototype nodes logged the in-vehicle CAN traffic of the Honda Accord 2013, and ran CIDS on messages 0x1B0, 0x1D0, 0x1A6, 0x294, 0x295, and 0x309. The approach in [32] was adopted to verify that messages {0x1B0, 0x1D0} were sent from the same ECU, {0x294, 0x295} both sent from another ECU, whereas others were sent from different ECUs. Utilizing these facts, one can conclude from Fig. 5(b) that the clock offsets and the skews derived in CIDS are equivalent *only* for those messages sent from the same ECU; 0x1B0 and 0x1D0 showed a skew of 78.4ppm, 0x294 and 0x295 showed a skew of 199.8ppm, while messages 0x1A6 and 0x309 showed very different skews of 265.7ppm and 95.78ppm, respectively. This result again shows that clock skews between *different* ECUs are distinct and can thus be used as the fingerprints of the corresponding ECUs.

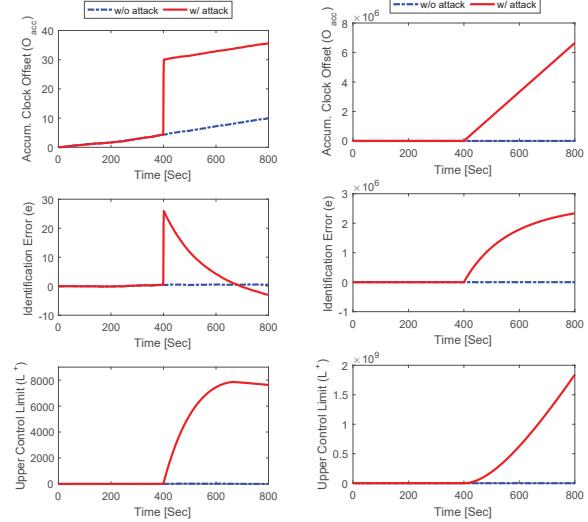
Toyota Camry 2010. To show that the applicability of CIDS’s fingerprinting is not limited to the specific vehicle model used, we also conducted experiments on a different vehicle: running CIDS’s fingerprinting on the Toyota Camry logged data. Similarly to the real vehicle evaluation in Section 5.1, the approach in [32] was used as the ground truth. It was verified that messages {0x20, 0xB2} within the CAN log data were all sent from some ECU A. Also, {0x223, 0x224} were both sent from some ECU B, whereas 0x2C1, 0x2C4, 0x3A0, 0x4C3, and 0x620 were each sent from a different ECU. As shown in Fig. 5(c), messages 0x20 and 0xB2 both showed a

clock skew of approximately 345.3ppm, whereas 0x223 and 0x224 showed a different clock skew of 276.5ppm. 0x2C4, 0x3A0, 0x4C3, and 0x620 showed very different clock skews of 460.1ppm, 142.5ppm, 26.1ppm and 58.7 ppm, respectively.

We made an interesting observation on message 0x2C1, showing a clock skew of 334.1ppm, which was different from the skews of messages {0x20, 0xB2} only by 3%, despite the fact that it was sent by a different ECU. This may confuse CIDS in determining whether they were sent by the same ECU or not. However, in such a case, CIDS can further examine the correlation between clock offsets and can thus fingerprint with a higher accuracy, which we will discuss and evaluate further in Section 5.4.

Dodge Ram Pickup 2010. We also ran CIDS’s fingerprinting on the CAN log data of a Dodge Ram Pickup 2010. For this vehicle, it was verified that message 0x200 was sent from some ECU A, {0x215, 0x300} sent from B, {0x6F9, 0x3E6, 0x6FD, 0x700} sent from C, and {0x101, 0x6FE} sent from D. Fig. 5(d) shows that CIDS determined that 0x200 has a clock skew of 351.7ppm, {0x215, 0x300} to have approximately 295.3ppm, {0x6F9, 0x3E6, 0x6FD, 0x700} to have 24.5ppm, and {0x101, 0x6FE} to have 110.3ppm, thus correctly fingerprinting their transmitters.

These results of a Toyota Camry and a Dodge Ram Pickup CAN log data again affirm the fact that the clock skews derived by CIDS are diverse and can indeed be used as fingerprints of in-vehicle ECUs. Moreover, they



(a) Fabrication attack.

(b) Suspension attack.

Figure 6: CIDS defending fabrication attack (left) and suspension attack (right) in a CAN bus prototype.

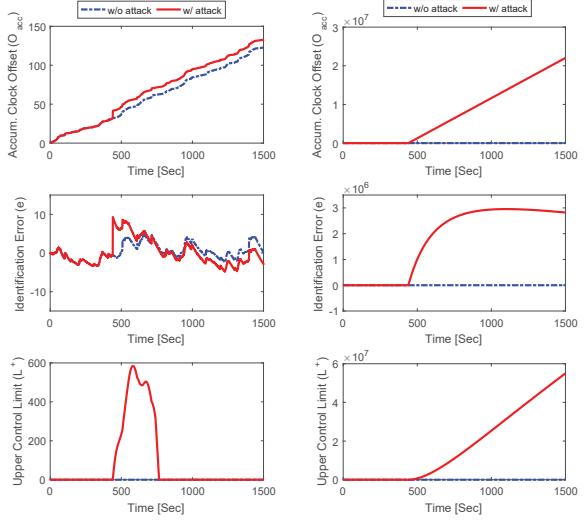
show that CIDS’s fingerprinting is not limited to a specific vehicle model, and can thus be applied to other vehicle models.

5.2 Defending Against Fabrication and Suspension Attacks

On both the CAN bus prototype and the real vehicle setting (Honda Accord 2013), we launch the fabrication and suspension attacks, and evaluate CIDS’s effectiveness in detecting them.² To this end, we consider CIDS to only perform per-message detection, and will later evaluate CIDS with message-pairwise detection.

CAN bus prototype. For evaluation of CIDS defending against fabrication attack on the CAN bus prototype, \mathbb{B} was programmed to inject a fabricated message at $t = 400$ secs with ID=0x11, which is a periodic message usually sent by \mathbb{A} , i.e., \mathbb{B} launches a fabrication attack on \mathbb{A} . ECU \mathbb{R} was running CIDS on message 0x11 and derived accumulated clock offset (O_{acc}), identification error (e), and control limits (L^+, L^-). For the suspension attack, \mathbb{A} was instead programmed to stop transmitting 0x11 at $t = 400$ secs.

Fig. 6(a) shows how such values changed for message 0x11 in the presence and absence of a fabrication attack. As soon as \mathbb{B} mounted a fabrication attack, as discussed in Section 4.2, there was a sudden positive shift in the accumulated clock offset, thus yielding a high iden-



(a) Fabrication attack.

(b) Suspension attack.

Figure 7: CIDS defending fabrication attack (left) and suspension attack (right) in a Honda Accord 2013.

tification error. Due to such a shift, the upper control limit, L^+ , of CUSUM suddenly increased and exceeded its threshold $\Gamma_L = 5$, i.e., detecting an intrusion. Similarly, Fig. 6(b) shows that since the suspension attack also shifted the accumulated clock offset significantly, CIDS was able to detect the attack.

Real vehicle. To evaluate CIDS against the fabrication attack under the real vehicle setting, one CAN prototype node \mathbb{R} was programmed to run CIDS, and another node \mathbb{A} as an adversary mounting the attack on a real ECU. The attack was mounted by injecting a fabricated attack message with ID=0x1B0, which was sent every 20ms by some real in-vehicle ECU, i.e., \mathbb{A} mounted the fabrication attack on a Honda Accord ECU sending 0x1B0. For the suspension attack, the message filter of \mathbb{R} was reset at $t = 420$ secs so as to no longer receive 0x1B0, thus emulating the suspension attack.

Fig. 7(a) shows how accumulated clock offsets (O_{acc}), identification errors (e), and upper control limits (L^+) changed for both cases of with and without a fabrication attack. Again, the attack message injected at around $t = 420$ secs caused a sudden increase in O_{acc} and e , thus increasing L^+ to exceed $\Gamma_L = 5$. As a result, CIDS declares the detection of an attack. After the attack, since 0x1B0 was still periodically sent by the real in-vehicle ECU, the clock skew — i.e., the slope of O_{acc} graph — remains unchanged. Similarly, as shown in Fig. 7(b), the suspension attack increases the offset values, thus causing L^+ to exceed the threshold, i.e., the suspension attack was detected by CIDS.

²As the attacks cannot be emulated using the CAN log data, we do not consider their use for evaluating CIDS against the attacks.

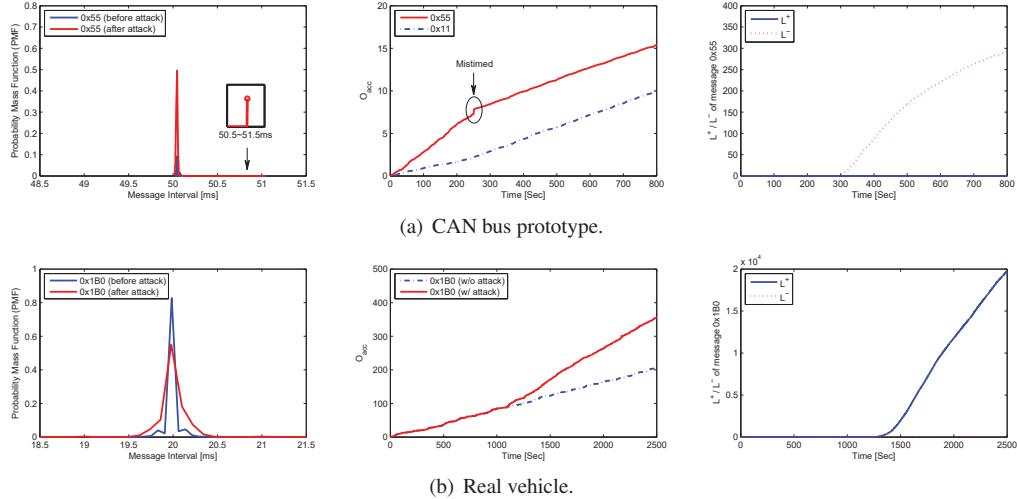


Figure 8: Masquerade attack — Probability mass function of message intervals (left), changes in accumulated clock offsets (middle), and control limits (right) derived in CIDS.

5.3 Defending Against Masquerade Attack

We now evaluate the performance of CIDS in detecting a masquerade attack.

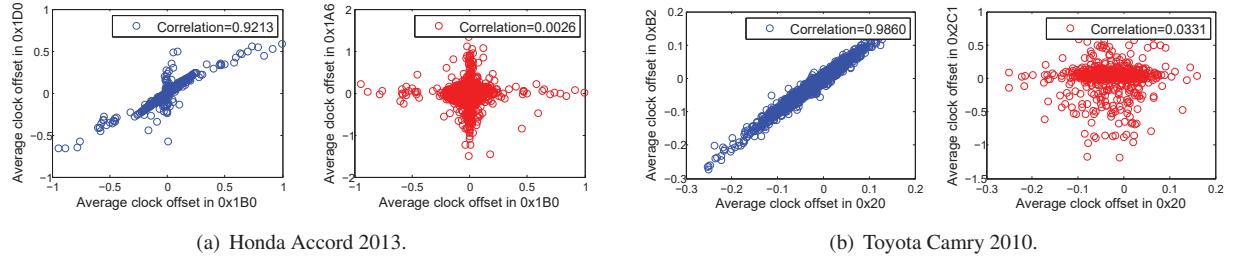
CAN bus prototype. To evaluate CIDS’s defense against the masquerade attack in the CAN bus prototype, nodes \mathbb{A} and \mathbb{B} were considered to have been compromised as strong and weak attackers as in Fig. 2(c), respectively. \mathbb{A} was programmed to mount a masquerade attack on \mathbb{B} , i.e., stop \mathbb{B} transmitting message 0x55 and instead send it through \mathbb{A} onwards, once $T_{masq} = 250$ secs had elapsed. As usual, messages 0x11 and 0x13 were periodically sent by \mathbb{A} , and CIDS was run by \mathbb{R} .

Fig. 8(a) (left) shows the Probability Mass Function (PMF) of the intervals of message 0x55: before and after the attack was mounted. In contrast to the fabrication attack, since the attacker sent the attack message at its original frequency after masquerading, the distribution did not deviate much from that before the attack. However, at T_{masq} , since there was some delay when the transmitter was switched from one node to another, the first masquerade attack message was sent 51.04ms after its previous transmission, whereas it should have been approximately 50ms which is the preset message interval of 0x55. Due to such a slightly *mistimed* masquerade attack, the PMF graph shows a message interval with an abnormal deviation from the mean. We will later evaluate the perfectly *timed* masquerade attack — a much more severe case than a mistimed attack — on a real vehicle, and show the efficacy of CIDS in detecting it.

The resulting changes in O_{acc} , L^+ , and L^- at \mathbb{R} are also shown in Fig. 8(a) (middle and right). The change in the ECU transmitting message 0x55 caused the slope

(i.e., clock skew) in O_{acc} graph to change after the attack was mounted. Since the measurements of O_{acc} after T_{masq} significantly deviated from their expected values, which is determined by the estimated clock skew of $t < T_{masq}$, the CUSUM lower control limit, L^- , in CIDS exceeded the threshold, thus declaring detection of an intrusion. Since the transmitter of 0x55 was changed (to ECU \mathbb{A}), its clock skew after $t = T_{masq}$ was equivalent to the clock skew in 0x11. Accordingly, via root-cause analysis, CIDS identifies the compromised ECU to be ECU \mathbb{A} . Unlike the previous results, since the change in slope was negative, persistent identification error with high negative values caused L^- to exceed the threshold.

Real vehicle. To evaluate CIDS’s defense against the masquerade attack in a real vehicle, we consider a scenario in which real in-vehicle ECUs \mathbb{V}_1 and \mathbb{V}_2 transmitting 0x1A6 and 0x1B0 are compromised as a strong and a weak attacker, respectively. Of the three CAN prototype nodes (\mathbb{A} , \mathbb{B} , and \mathbb{R}), which were connected to the real in-vehicle network via OBD-II, we programmed node \mathbb{R} to run CIDS on in-vehicle message 0x1B0 and another node \mathbb{B} to simply log the CAN traffic. To generate a scenario of real ECU \mathbb{V}_1 mounting a masquerade attack on real ECU \mathbb{V}_2 , \mathbb{R} was programmed further to receive message 0x1A6 instead of 0x1B0, but still record the received messages’ ID to be 0x1B0, once $T_{masq} = 1100$ seconds had elapsed. That is, we let \mathbb{R} interpret 0x1A6 as 0x1B0 for $t > T_{masq}$, i.e., the transmitter of 0x1B0 changes from \mathbb{V}_2 to \mathbb{V}_1 . Such a change in interpretation was achieved by programming \mathbb{R} to modify its message acceptance filter from only accepting 0x1B0 to only accepting 0x1A6. Since 0x1B0 and 0x1A6 were observed to be always transmitted nearly at the same time, such a



(a) Honda Accord 2013.

(b) Toyota Camry 2010.

Figure 9: Correlated and uncorrelated clock offsets.

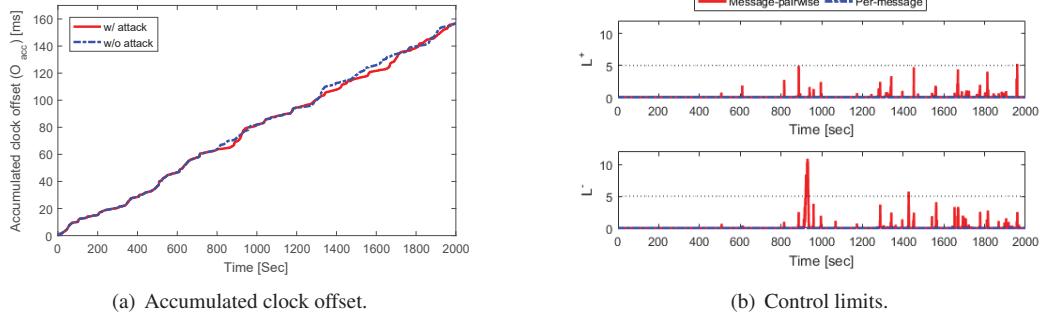


Figure 10: Defense against the worst-case masquerade attack via message-pairwise detection.

setting replicates the *timed* masquerade attack. During such a process, \mathbb{B} continuously logged 0x1B0 so that we can obtain a reference for circumstances when no attacks are mounted.

Fig. 8(b) (left) shows the PMF of the message intervals of 0x1B0 before and after the attack. Since the message periodicity remained the same, the distribution of the messages intervals did not change. Moreover, since we considered a timed masquerade attack, in contrast to the result in Fig. 8(a), there were no such abnormal message intervals. Such a result indicates that state-of-the-art IDSs, which try to find abnormal message frequencies, cannot detect such an attack. Although the distribution of message intervals remained unchanged, due to the change in ECU transmitting 0x1B0 ($V_2 \rightarrow V_1$), the accumulated clock offset suddenly exhibited a different trend in its change, i.e., a different clock skew after the attack. Here, the original trend in offset changes was determined by the data obtained from \mathbb{B} . So, as shown in Fig. 8(b) (right), CIDS was able to detect a sudden shift in its identification error and thus outputted a high level of CUSUM upper control limit, i.e., an intrusion detection. CIDS's capability of detecting various types of masquerade attack is evaluated further in Section 5.5.

In conclusion, through its modeling and detection processes, CIDS can detect not only the fabrication attack but also the masquerade attack, i.e., is capable of doing not only what existing solutions can do, but also more.

5.4 Message-pairwise Detection

We evaluate the feasibility and efficiency of message-pairwise detection in CIDS. To validate its practicability in the real-world, we first examine whether there exists pairs of messages inside real vehicles with correlated clock offsets — the condition for CIDS to run message-pairwise detection.

Fig. 9(a) shows two cases of correlated and uncorrelated clock offsets of in-vehicle messages collected from the Honda Accord 2013. Fig. 9(a) (left) shows that the average clock offsets of messages 0x1B0 and 0x1D0, which were determined to have been sent from the same ECU, showed a high correlation of 0.9213, i.e., linear relationship. In contrast, as shown in Fig. 9(a) (right), average clock offsets of messages 0x1B0 and 0x1A6, which were sent every 20ms from different ECUs, showed a near 0 correlation.

By the Birthday paradox, some ECUs in the vehicle may probably have near-equivalent clock skews — as it was for messages 0x20 and 0x2C1 in the examined Toyota Camry 2010 (see Fig. 5(c)). Although clock skews may be near-equivalent, instantaneous clock offsets of two different ECUs cannot be near-equivalent and are thus uncorrelated as they run different processes. The results in Fig. 9(b) corroborate such a fact by showing that clock offsets of messages 0x20 and 0xB2, which were sent by the same ECU, had a high correlation of 0.9860, whereas offsets of messages 0x20 and 0x2C1 —

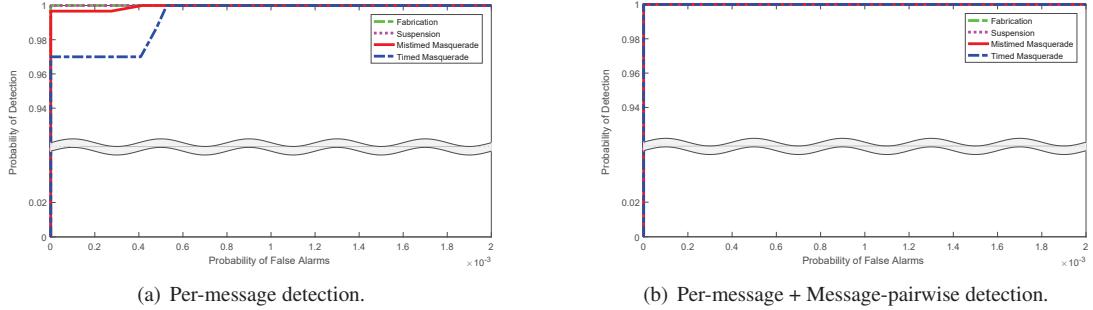


Figure 11: ROC curves of CIDS in the real vehicle.

sent by different ECUs with similar clock skews — had a low correlation of 0.0331. Thus, for messages with near-equivalent clock skews, CIDS can further examine the correlation between their clock offsets, and correctly determine their transmitters.³ These facts and observations indicate the feasibility and efficiency of message-pairwise detection in CIDS.

To show that message-pairwise detection can support per-message detection in decreasing false positives/negatives by examining offset correlations, we consider a scenario in which an attacker \mathbb{V}_1 has mounted a masquerade attack on a Honda Accord ECU \mathbb{V}_2 at $t_{masq} = 800$ secs. We refer to \mathbb{V}_2 as the ECU which originally transmits message 0x1B0. To consider the *worst case* in detecting the masquerade attack, we assume that the clock skews of \mathbb{V}_1 and \mathbb{V}_2 are nearly equivalent, similarly to messages 0x20 and 0x2C1 in the Toyota Camry. We replicated such a worst-case scenario by randomly permuting the acquired offset values of 0x1B0 for $t > t_{masq}$, and considering the permuted values to be output from \mathbb{V}_1 . As shown in Fig. 10(a), this leads to a situation where the clock skew does not change even though the message transmitter has been changed from one ECU to another. Although the clock skew remained equivalent at $t = t_{masq}$, the correlation between offsets of 0x1B0 and 0x1D0 suddenly dropped from 0.9533 to 0.1201, i.e., a linear to non-linear relationship. As a result, as shown in Fig. 10(b), the control limits in CIDS’s message-pairwise detection exceeded the threshold $\Gamma_L = 5$. On the other hand, since the clock skews before and after the attack were equivalent, per-message detection was not able to detect the intrusion.

5.5 False Alarm Rate

We also examined the false alarm rate of CIDS under the real vehicle setting. The results obtained from the CAN

³If the two ECUs’ clock behaviors are still not distinguishable, CIDS can be set up to exclude them for examination so that the risk of false positives significantly decreases. However, this may impact CIDS’s capability of detecting attacks mounted through those ECUs.

bus prototype are omitted due to their insignificance, i.e., not many false alarms occurred due to its less complex bus traffic. Based on data recorded for 30 minutes from the Honda Accord 2013 — approximately 2.25 million messages on the CAN bus — four attack datasets were constructed to each contain 300 different intrusions. The intrusions either had different injection timings, suspension timings, or changes in clock skews: each in the form of fabrication attack, suspension attack, mistimed masquerade attack, and timed masquerade attack. For each dataset, we varied the κ parameter of CIDS to acquire one false positive rate (false-alarm rate) and one false negative rate (1–detection rate).

Fig. 11(a) shows the Receiver Operating Characteristic (ROC) curve of CIDS, which represents its trade-off between false alarm and detection, executing *only* per-message detection on the attack datasets. Clearly, CIDS is shown to be able to detect fabrication, suspension, and masquerade attacks with a high probability. Since the timed masquerade attack is the most difficult to detect, it showed the highest false positive rate among all the attack scenarios considered: a false positive rate of 0.055% while not missing any anomalies (100% true positives). Even for false positives $< 0.055\%$, 97% of the anomalies were detected by CIDS. However, these false positives can be of great concern for in-vehicle networks. Therefore, to eliminate such false positives, CIDS can additionally run message-pairwise detection. Fig. 11(b) shows the ROC curve of CIDS executing not only per-message detection but also message-pairwise detection for further verification. Accordingly, CIDS was able to detect all types of attacks considered without having any false positives, which is in contrast to CIDS with only per-message detection, i.e., all false positives were eliminated via message-pairwise detection.

6 Discussion

Discussed below are the overhead, deployment, limitations, and applications of CIDS.

Identification algorithm. To estimate clock skew, one can also use other algorithms than RLS, such as Total Least Squares (TLS) and Damped Least Squares (DLS), which perform orthogonal linear and non-linear regression, respectively. Although they might identify the clock skew with a higher accuracy than RLS, their gains are offset by the accompanying high complexity. TLS requires Singular Value Decomposition (SVD), which is computationally expensive, and DLS requires a large number of iterations for curve fitting. RLS is known to have a computation complexity of $\mathcal{O}(N^2)$ per iteration, where N is the size of the data matrix. However, in CIDS, only a scalar clock offset is exploited for identification, and thus the computational complexity is relatively low.

Defeating CIDS. There may be several ways the adversary may attempt to defeat CIDS. First, the adversary may try to compromise the ECU running CIDS and disable it. However, if *cross-validation* for CIDS was to be exploited, such an attempt can be nullified. For the detection of intrusions, CIDS only requires an ECU to record the timestamps of message arrivals. Such a low overhead makes it feasible for CIDS to be installed distributively across several in-vehicle ECUs for cross-validation. Suppose using CIDS, ECU A monitors attacks on messages $\{M_1, M_2\}$, ECU B monitors $\{M_2, M_3\}$, and ECU C monitors $\{M_1, M_3\}$. Since CIDS regards the *receiver's* time clock as the true clock, cross-validation provides multiple perspectives of clock behaviors for each message ID, e.g., two different perspectives of M_2 from A and B. Thus, even when an ECU running CIDS gets compromised, cross-validation via CIDS can handle such a problem.

Another way the adversary may try to defeat CIDS is to adapt to how its algorithm is running and thus deceive it. The adversary may figure out the clock skew of the target ECU and then heat up or cool down the compromised ECU so that its clock skew changes to match that of the target. In such a case, the clock skew can be matched and thus may bypass CIDS's per-message detection. However, as discussed in Section 5.4, unless the adversary also matches the instantaneous clock offset, which is affected by the ECU's *momentary* workload and temperature, CIDS can detect the intrusion via message-pairwise detection.

Upon intrusion detection. False alarms for intrusion detection systems, especially in in-vehicle networks, are critical. Thus, CIDS should also deal with them as accurately as possible. To meet this requirement, if an intrusion has been determined, even after going through the verification process, CIDS can follow the following steps for further examination:

1. If an intrusion was detected while using only per-message detection, examine it further via message-pairwise detection.

2. If still alarmed as an intrusion and the attacked ECU is a safety-critical ECU, go straight to step 4.
3. If not, communicate with other ECUs for cross-validation as they would provide different perspectives of the clock skew results. If communicating with other ECUs incurs too much overhead (in terms of bus load, processing overhead, etc.), send traffic data for a remote diagnosis.
4. Request re-patching of firmware and advise the driver to stop the vehicle.

Limitation of CIDS. CIDS is shown to be effective in detecting various types of in-vehicle network intrusions. One limitation of CIDS might be that since it can only extract clock skews from periodic messages, it would be difficult to fingerprint ECUs which are sending aperiodic messages. That is, if the attacker injects messages aperiodically, although CIDS can still detect the intrusion, it would not be able to pinpoint where the attack message came from, i.e., finding the root-cause of attacks launched with or on aperiodic messages. Recall that CIDS can achieve this only for periodic messages. In future, we would like to find new features other than clock skew, which can fingerprint ECUs, regardless of whether they send messages periodically or aperiodically.

Applicability to other in-vehicle networks. Although most modern in-vehicle networks are based on CAN, some may be equipped with other protocols, such as CAN-FD, TTCAN and FlexRay, for more complex operations. CAN-FD is an enhanced version of CAN, providing flexible and higher data rates [5]. Since its basic components conform to CAN and thus also lacks synchronization, CIDS can be applied to CAN-FD. For protocols such as TTCAN [21] and FlexRay [22], nodes are periodically synchronized for determinative timing of message exchanges. The interval between two consecutive synchronizations depends on how each protocol is deployed [32]. For TTCAN, it can be up to $2^{16} = 65536$ bits long, i.e., 131ms in a 500Kbps bus [37]. This lets some messages be sent multiple times between consecutive synchronizations. So, if the time interval is long, CIDS would still be able to extract clock skews from messages which are sent multiple times, whereas, if the period is short, CIDS may not be feasible. However, the fact that TTCAN and FlexRay have high implementation cost, whereas for CAN-FD it is minimal, makes CAN-FD a favorite candidate for next-generation in-vehicle networks [6, 39]. This means that CIDS can be applicable to not only current but also future in-vehicle networks.

7 Conclusion

New security breaches in vehicles have made vehicle security one of the most critical issues. To defend against

vehicle attacks, several security mechanisms have been proposed in the literature. They can cope with some attacks but cannot cover other safety-critical attacks, such as the masquerade attack. To remedy this problem, we have proposed a new IDS called CIDS, which extracts clock skews from message intervals, fingerprints the transmitter ECUs, and models their clock behaviors using RLS. Then, based on the thus-constructed model, CIDS detects intrusions via CUSUM analysis. Based on our experiments on a CAN bus prototype and on real vehicles, CIDS is shown to be capable of detecting various types of in-vehicle network intrusions. CIDS can address all attacks that existing IDSs can and cannot handle as well as facilitates root-cause analysis. Thus, it has potential for significantly enhancing vehicle security and safety.

8 Acknowledgments

We would like to thank the anonymous reviewers and the shepherd, Tadayoshi Kohno, for constructive suggestions. The work reported in this paper was supported in part by the NSF/Intel Grant CNS-1505785 and the DGIST Global Research Laboratory Program through NRF funded by MSIP of Korea (2013K1A1A2A02078326).

References

- [1] Microchip MCP2515 Datasheet. [Online] Available: www.microchip.com/MCP2515.
- [2] Microchip TB078, PLL Jitter and Its Effects in the CAN Protocol.
- [3] On-Board Diagnostic System [Online] <http://www.obdii.com>.
- [4] CAN Specification v2.0. *Robert Bosch GmbH* (1991).
- [5] CAN with Flexible Data-Rate Specification Version 1.0. *Robert Bosch GmbH* (2012).
- [6] Infineon: CAN FD Success Goes at Expense of FlexRay. [Online] <http://www.eetimes.com/>. *EETimes* (Feb. 2015).
- [7] Hackers Remotely Kill a Jeep on the Highway - With Me in It. [Online] <http://www.wired.com>.
- [8] BASSEVILLE, M., AND NIKIFOROV, I. Detection of abrupt changes: Theory and application. In *Prentice Hall information and system sciences series* (1993).
- [9] CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., SAVAGE, S., KOSCHER, K., CZEKSIS, A., ROESNER, F., AND KOHNO, T. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *USENIX Security* (2011).
- [10] CHO, K. T., PARK, T., AND SHIN, K. G. CPS Approach to Checking Norm Operation of a Brake-by-Wire System. In *ICCPS* (2015).
- [11] DAILY, J. Analysis of critical speed yaw scuffs using spiral curves. In *SAE Technical Paper 2012-01-0606* (2012).
- [12] DAVIS, R., KOLLMANN, S., POLLEX, V., AND SLOMKA, F. Controller area network (can) schedulability analysis with fifo queues. *ECRTS* (2011).
- [13] FOSTER, I., PRUDHOMME, A., KOSCHER, K., AND SAVAGE, S. Fast and Vulnerable: A Story of Telematic Failures. In *WOOT* (2015).
- [14] HAYKIN, S. Adaptive filter theory. In *2nd ed. Prentice-Hall* (1991).
- [15] HERREWEGE, A., SINGELLEE, D., AND VERBAUWHEDE, I. Canauth - a simple, backward compatible broadcast authentication protocol for can bus. In *ECRYPT Workshop on Lightweight Cryptography* (2011).
- [16] HOPPE, T., KILTZ, S., AND DITTMANN, J. Security threats to automotive can networks - practical examples and selected short-term countermeasures. In *Reliability Engineering and System Safety* (Jan. 2011).
- [17] JANA, S., AND KASERA, S. K. On fast and accurate detection of unauthorized wireless access points using clock skews. In *ACM MobiCom* (2008).
- [18] KHAN, D., BRIL, R., AND NAVET, N. Integrating hardware limitations in can schedulability analysis. In *WFCS* (May. 2010).
- [19] KOHNO, T., BRODIO, A., AND CLAFFY, K. Remote physical device fingerprinting. In *IEEE Symposium on Security and Privacy* (2005).
- [20] KOSCHER, K., CZEKSIS, A., ROESNER, F., PATEL, S., KOHNO, T., CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., AND SAVAGE, S. Experimental security analysis of a modern automobile. In *IEEE Security and Privacy* (2010).
- [21] LEEN, G., AND HEFFERNAN, D. Ttcan: a new time-triggered controller area network. In *Elsevier Microprocessors and Microsystems* (2002).
- [22] MILBREDT, P., HORAUER, M., AND STEININGER, A. An investigation of the clique problem in flexray. *SIES* (2008).
- [23] MILLER, C., AND VALASEK, C. Adventures in automotive networks and control units. *Defcon 21* (2013).
- [24] MILLER, C., AND VALASEK, C. A survey of remote automotive attack surfaces. *Black Hat USA* (2014).
- [25] MILLER, C., AND VALASEK, C. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* (2015).
- [26] MILLS, D. L. Network time protocol: Specification, implementation, and analysis. RFC 1305.
- [27] MOHALIK, S., RAJEEV, A. C., DIXIT, M. G., RAMESH, S., SUMAN, P. V., PANDYA, P. K., AND JIANG, S. Model checking based analysis of end-to-end latency in embedded, real-time systems with clock drifts. In *DAC* (2008).
- [28] MONTGOMERY, D. Introduction to statistical quality control. In *4th edition, Wiley* (2000).
- [29] MOON, S. B., SKELLY, P., AND TOWSLEY, D. Estimation and removal of clock skew from network delay measurements. In *INFOCOM* (1999).
- [30] MUTER, M., AND ASAI, N. Entropy-based anomaly detection for in-vehicle networks. *IEEE IVS* (2011).
- [31] MUTER, M., GROLL, A., , AND FREILING, F. C. A structured approach to anomaly detection for in-vehicle networks. In *Information Assurance and Security (IAS), Sixth International Conference* (2010).
- [32] NATALE, M. D., ZENG, H., GIUSTO, P., AND GHOSAL, A. Understanding and using the controller area network communication protocol: Theory and practice. In *Springer Science & Business Media* (2012).
- [33] NILSSON, D., LARSON, D., AND JONSSON, E. Efficient In-Vehicle Delayed Data Authentication Based on Compound Message Authentication Codes. In *VTC-Fall* (2008).

- [34] PASZTOR, A., AND VEITCH, D. Pc based precision timing without gps. In *ACM SIGMETRICS* (2002).
- [35] PAXSON, V. On calibrating measurements of packet transit times. In *SIGMETRICS* (1998).
- [36] RUTH, R., BARTLETT, W., AND DAILY, J. Accuracy of event data in the 2010 and 2011 Toyota camry during steady state and braking conditions. In *SAE International Journal on Passenger Cars* (2012).
- [37] RYANA, C., HEFFERNANB, D., AND LEENA, G. Clock synchronisation on multiple ttcn network channels. In *Elsevier Microprocessors and Microsystems* (2004).
- [38] SZILAGYI, C., AND KOOPMAN, P. Low cost multicast network authentication for embedded control systems. In *Proceedings of the 5th Workshop on Embedded Systems Security* (2010).
- [39] TALBOT, S. C., AND REN, S. Comparison of fieldbus systems, can, ttcn, flexray and lin in passenger vehicles. In *ICDCSW* (2009).
- [40] VEITCH, D., BABU, S., AND PASZTOR, A. Robust synchronization of software cclock across the internet. In *IMC* (2004).
- [41] WOODALL, W. H., AND ADAMS, B. The statistical design of cusum charts. In *Quality Engineering*, 5(4), (1993).
- [42] ZANDER, S., AND MURDOCH, S. An improved clock-skew measurement technique for revealing hidden services. In *USENIX Security* (2008).

Lock It and Still Lose It – On the (In)Security of Automotive Remote Keyless Entry Systems

Flavio D. Garcia¹

*School of Computer Science,
University of Birmingham, UK.
f.garcia@bham.ac.uk*

Timo Kasper²

*Kasper & Oswald GmbH, Germany.
info@kasper-oswald.de*

David Oswald²

*School of Computer Science,
University of Birmingham, UK.
d.oswald@bham.ac.uk*

Pierre Pavlidès¹

*School of Computer Science,
University of Birmingham, UK.
pierre@pavlides.fr*

Abstract

While most automotive immobilizer systems have been shown to be insecure in the last few years, the security of remote keyless entry systems (to lock and unlock a car) based on rolling codes has received less attention. In this paper, we close this gap and present vulnerabilities in keyless entry schemes used by major manufacturers. In our first case study, we show that the security of the keyless entry systems of most VW Group vehicles manufactured between 1995 and today relies on a few, global master keys. We show that by recovering the cryptographic algorithms and keys from electronic control units, an adversary is able to clone a VW Group remote control and gain unauthorized access to a vehicle by eavesdropping a single signal sent by the original remote. Secondly, we describe the Hitag2 rolling code scheme (used in vehicles made by Alfa Romeo, Chevrolet, Peugeot, Lancia, Opel, Renault, and Ford among others) in full detail. We present a novel correlation-based attack on Hitag2, which allows recovery of the cryptographic key and thus cloning of the remote control with four to eight rolling codes and a few minutes of computation on a laptop. Our findings affect millions of vehicles worldwide and could explain unsolved insurance cases of theft from allegedly locked vehicles.

1 Car Keys

For several decades, car keys have been used to physically secure vehicles. Initially, simple mechanical keys were introduced to open the doors, unlock the steering, and operate the ignition lock to start the engine. Given physical access to a mechanical key, or at hand of a detailed photograph, it is possible

to create a duplicate. In addition, mechanical tumbler locks and disc locks are known to be vulnerable to techniques such as lock-picking and bumping that allow to operate a lock without the respective key. Finally, for most types of car locks, locksmith tools exist that allow to decode the lock and create a matching key.

1.1 Electronics in a Car Key

With electronic accessories becoming available, additional features were integrated into the locking and starting systems of cars: some of them to improve the comfort, others to increase security. On the side of the car key, this implies some electronic circuitry integrated in its plastic shell, as illustrated in Figure 1. Note that the link between Remote Keyless Entry (RKE) and immobilizer is optional. In the Hitag2 system (Section 4), the immobilizer interface can be used to re-synchronize the counter used for RKE, while VW Group vehicles (Section 3) completely separate RKE and immobilizer. In vehicles with Passive Keyless Entry and Start (PKES) (Section 1.1.2), the low-frequency immobilizer link is used to trigger the transmission of a door opening signal over the high-frequency RKE interface.

1.1.1 Immobilizer Transponders

One of the most notable events in the history of car security was the introduction of the immobilizer, which significantly reduced the number of stolen cars and so-called joyrides conducted by teenagers. An electronic immobilizer improves the security of the car key with respect to starting the engine. Technically, most immobilizers rely on Radio Frequency IDentification (RFID) technology: An RFID transponder is embedded in the plastic shell of the car key and contains a secret that is required to

¹These authors contributed the research on Hitag2.

²These authors contributed the research on VW Group.

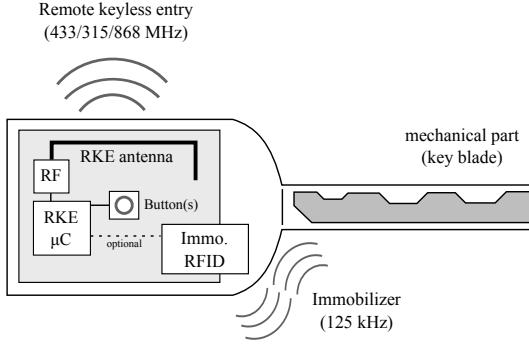


Figure 1: Main components of a car key: RKE and immobilizer systems are separate and use different RF frequencies.

switch on the ignition and start the engine. An antenna coil around the ignition lock establishes a bidirectional communication link and provides the energy for the transponder in order to verify its authenticity with a range of a few centimeters. All modern immobilizers use cryptography for authentication between transponder and vehicle, typically based on a challenge-response protocol.

For many years, only weak, proprietary cryptography was implemented in immobilizer transponders worldwide. This may have been caused by the limited energy available on RFID-powered devices, technological limitations, and cost considerations. The first type of immobilizer transponder to be broken was the widespread DST40 cipher used in Texas Instrument’s Digital Signature Transponder (DST), which was reverse-engineered and broken at Usenix Security 2005 [8]: The 40-bit secret key of the cipher can be revealed in a short time by means of exhaustive search. This paper was at the same time one of the first published attacks on a commercial device in the literature. A few years later, at Usenix Security 2012, researchers published several cryptanalytic attacks on NXP’s Hitag2 transponders [30, 32], the most widely used car immobilizer at that time. The authors showed that an attacker can obtain the 48-bit secret key required to bypass the electronic protection in less than 360 seconds. One year later, in a paper submitted to Usenix Security 2013 (and finally published in 2015), the security mechanism of the Megamos Crypto transponder were found to be vulnerable to cryptanalytic attacks [31, 33]. The 96-bit secret key of the cipher is mapped into a 57-bit state of a stream cipher that can be rolled back. A flawed key generation (multiple bits of the secret key are set to zero) additionally found in various transponders decreases the attack time from the order of days to a few seconds

using a Time-Memory Tradeoff (TMTO).

As a result, the majority of RFID immobilizers used in today’s vehicles can be cloned: the secret of the transponder can be obtained by an adversary to circumvent the added security provided by the immobilizer. The cryptography of these immobilizers has to be considered broken as their added protection to prevent criminals from starting the engine of a car is very weak.

1.1.2 Passive Keyless Entry and Start

Today, certain modern cars (especially made by luxury brands) are equipped with PKES systems that rely on a bidirectional challenge-response scheme, with a small operating range of about one meter: When in proximity of the vehicle, the car key generates a cryptographic response to a challenge transmitted by the car. A valid response unlocks the doors, deactivates the alarm system, and enables the engine to start. As a consequence, the only remaining mechanical part in some cars is a door lock for emergencies (usually found behind a plastic cover on the driver’s side), to be used when the battery is depleted.

Unfortunately, PKES does not require user interaction (such as a button press) on the side of the car key to initiate the cryptographic computations and signal transmission. The lack of user interaction makes PKES systems prone to relay attacks, in which the challenge and response signals are relayed via a separate wireless channel: The car key (e.g., in the pocket of the victim) and vehicle (e.g., parked hundreds of meters away) will assume their mutual proximity and successfully authenticate. Since the initial publication of these relay attacks in 2011 [14], tools that automatically perform relay attacks on PKES systems are available on the black market and are potentially used by criminals to open, start, and steal vehicles.

1.1.3 Remote Keyless Entry Systems

RKE systems rely on a unidirectional data transmission from the remote control, which is embedded in the car key, to the vehicle. Upon pressing a button, an active Radio Frequency (RF) transmitter in the remote control usually generates signals in a freely usable frequency band. These include the 315 MHz band in North America and the 433 MHz or 868 MHz band in Europe, with a typical range of several tens to hundreds of meters. Note that a few old cars have been using infrared technology instead of RF. RKE systems enable the user to comfortably lock and un-

lock the vehicle from a distance, and can be used to switch on and off the anti-theft alarm, when present.

The first remote controls for cars used no cryptography at all: The car was unlocked after the successful reception of a constant “fix code” signal. Replay attacks on these systems are straightforward. We encountered a Mercedes Benz vehicle manufactured around 2000 that still relies on such fix code RKE systems.

The next generation of RKE systems are so-called rolling code systems, which employ cryptography and a counter value that is increased on each button press. The counter value (and other inputs) form the plaintext for generating a new, encrypted (or otherwise authenticated) rolling code signal. After decryption/verification on the side of the vehicle, the counter value is checked by comparing it to the last stored counter value that was recognized as valid: An increased counter value is considered new and thus accepted. A rolling code with an old counter value is rejected. This mechanism constitutes an effective protection against replay attacks, since a rolling code is invalidated once it has been received by the vehicle. The cryptographic mechanisms behind rolling code systems are further described in Section 2.

In principle, such unidirectional rolling code schemes can provide a suitable security level for access control. However, as researchers have shown in the case of KEELOQ in 2008, the security guarantees are invalidated if they rely on flawed cryptographic schemes: KEELOQ was broken both by cryptanalysis [7, 15] and, in a more realistic setting, by side-channel attacks on the key derivation scheme executed by the receiver unit [12, 17]. Although it is frequently mentioned that KEELOQ is widely used for vehicle RKE systems, our research indicates that this system is prevalently employed for garage door openers.

Another attack, targeting an outdated automotive RKE scheme of an unspecified vehicle (built between 2000 and 2005), was demonstrated by Cesare in 2014 [9]: An adversary has to eavesdrop three subsequent rolling codes. Then, using phase-space analysis, the next rolling code can be predicted with a high probability. However, apart from this attack the cryptographic security of automotive RKE systems has not been investigated to our knowledge. In particular, a large-scale survey and security analysis of very wide-spread rolling code systems has not been carried out.

A different, simple but effective method used by criminals to break into cars is to jam the RF communication when the victim presses the remote con-

trol to lock the car. The victim may not notice the attack and thus leave the car open. A variant of the attack is “selective jamming”, i.e., a combined eavesdropping-and-jamming approach: The transmitted rolling code signal is monitored and at the same time jammed, with the effect that the car is not locked and the attacker possesses a temporarily valid (one-time) rolling code. Consequently, a car could be found appropriately locked after a burglary. This approach was first mentioned in [17] and later practically demonstrated by [16, 27]. Note that one successful transmission of a new rolling code from the original remote to the car usually invalidates all previously eavesdropped rolling codes, i.e., the time window for the attack is relatively small. Furthermore, it is usually not possible to change the signal contents, for example, convert a “lock” command into an “unlock”. This limitation is often overlooked (e.g. in [16, 27]) and severely limits the practical threat posed by this type of attack.

1.2 Contribution and Outline

In this paper, we study several extremely widespread RKE systems and reveal severe vulnerabilities, affecting millions of vehicles worldwide. Our research was in part motivated by reports of unexplained burglaries of locked vehicles (for example [1, 2]), as well as scientific curiosity regarding the security of our own, personal vehicles.

The remainder of this paper is structured as follows: In Section 2, we briefly summarize the results of our preliminary analysis of different RKE systems solely by analyzing the transmitted RF signals. The main contributions presented subsequently are:

1. In Section 3, we analyze the RKE schemes employed in most VW Group group vehicles between 1995 and today. By reverse-engineering the firmware of the respective Electronic Control Units (ECUs), we discovered that VW Group RKE systems rely on cryptographic schemes with a single, worldwide master key, which allows an adversary to gain unauthorized access to an affected vehicle after eavesdropping a single rolling code.
2. In Section 4, we study an RKE scheme based on the Hitag2 cipher, as used by many different manufacturers. We have reverse-engineered the protocol in a black-box fashion and present a novel, fast correlation attack on Hitag2 applicable in an RKE context. By eavesdropping four to eight rolling codes, an adversary can re-

cover the cryptographic key within minutes and afterwards clone the original remote control.

2 Preliminary Analysis of RKE Systems

To address the research question of this paper: “how secure are modern automotive RKE systems?”, we captured RF signals from the remote controls of a variety of vehicles, including our own cars (VW Passat 3B, Škoda Fabia, Alfa Romeo Giulietta). Today, the required hardware for receiving (and sending) RKE signals is widely available. For our analyses, we used various devices, including Software-Defined Radios (SDRs) (HackRF, USRP, `rtl-sdr` DVB-T USB sticks) and inexpensive RF modules. Figure 2 shows our simple setup which costs $\approx \$40$, is battery-powered, can eavesdrop and record rolling codes, emulate a key, and perform reactive jamming.

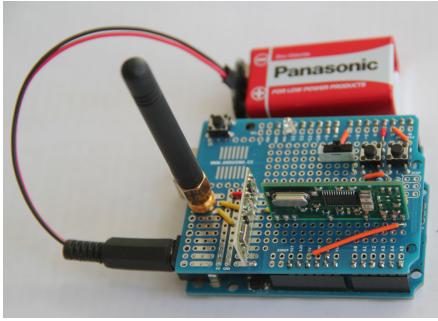


Figure 2: Arduino-based RF transceiver

Studying the raw received signals and guessing the respective modulation and encoding schemes turned out to be straightforward: The majority of the studied RKE systems uses simple Amplitude-Shift Keying (ASK) as modulation scheme, while a smaller percentage employs Frequency Shift Keying (FSK). For the encoding of the actual data bits, the most prevalent methods are Manchester encoding and pulse-width encoding. The utilized bit rates range from less than 1 kBit/s (for older remotes) to 20 kBit/s (for newer remotes).

A typical rolling code packet consists of a preamble (i.e., a regular sequence of 0 and 1), a fixed start pattern (a sequence of one or a few fixed bytes), the actual, cryptographic data payload, and a final checksum, cf. Figure 3. Note that many schemes slightly deviate from this general structure. Also, in virtually all cases, the same packet is sent multiple times, presumably to increase the reliability in presence of environmental disturbances.

The data payload normally contains the Unique

Preamble	Start pattern	Payload	Checksum
----------	---------------	---------	----------

Figure 3: General packet structure of a rolling code. Gray background indicates that the part is either encrypted or authenticated.

Identifier (UID) of the remote control, the rolling counter value, and the pressed button (i.e., “unlock”, “lock”, “open trunk”, in the US also “panic” or “alarm”). Obviously, the data sent by the remote control has to be cryptographically authenticated in some way. There appear to be two major routes that were taken by designers of RKE systems:

Implicit authentication: The complete payload (or part of it) is symmetrically encrypted. The receiver then decrypts the packet, and checks if the content is valid, i.e., if the UID is known to the vehicle and the counter is in its validity window. Examples for this approach can be found in Section 3.

Explicit authentication: Some form of Message Authentication Code (MAC) is computed over the data payload and then appended to the packet. An example of this approach is the Hitag2 scheme described in Section 4.

As a next step, we tried to determine the utilized encryption algorithms. However, a search for publicly available documentation or data sheets yielded little results. For example, the systems employed in VW Group vehicles (VW, Seat, Škoda, and Audi) appear to be a complete black box without any publicly documented security analysis. Since VW Group vehicles are extremely wide-spread, we selected this manufacturer as the target of our first case study (Section 3). Our second case study focuses on the Hitag2 scheme, for which abridged (one-page) data sheets can be found on the Internet [26]. We found Hitag2-based remote controls in vehicles made by a variety of manufacturers, hence, we opted to recover the exact functionality and further analyze the security of this RKE scheme (Section 4).

3 Case Study 1: The VW System

With over 23% market share in Europe (September 2015) and 11.1% worldwide (August 2014), the VW Group is amongst the leading global automotive manufacturers [13]. We had access to a wide variety of VW Group vehicles for our security analysis, from vehicles manufactured in the early 2000s to ones for the model year 2016. In total, the VW Group has sold almost 100 million cars from 2002 until 2015. While not all of these vehicles use the

RKE schemes covered in this section, we have strong indications that the vast majority is vulnerable to the attacks presented in the following. Note that the VW Group also includes certain luxury brands (e.g., Porsche, Bentley, Lamborghini, Bugatti) that we did not analyze in detail. Instead, we focused on more wide-spread vehicles manufactured by VW, Seat, Škoda, and Audi. For a list of cars that we validated our findings with, refer to Section 3.5.1. Eavesdropping and analyzing the signals transmitted by numerous remote controls, we identified at least 7 different RKE schemes, referred to as VW-x ($x = 1 \dots 7$) in the following. Out of these systems, we selected the four schemes covering the largest amount of vehicles:

VW-1: The oldest system, used in model years until approximately 2005. The remote control transmits On-Off-Keying (OOK) modulated signals at 433.92 MHz, using pulse-width coding at a bitrate of 0.667 kBit/s.

VW-2: Used from approximately 2004 onwards. The operating frequency is 434.4 MHz using OOK (same as for VW-3 and VW-4), transmitting Manchester-encoded data at a bitrate of 1 kBit/s.

VW-3: Employed for models from approximately 2006 onwards, using a frequency of 434.4 MHz and Manchester encoding at a bitrate of 1.667 kBit/s. The packet format differs considerably from VW-2.

VW-4: The most recent scheme we analyzed, found in vehicles between approximately 2009 and 2016. The system shares frequency, encoding, and packet format with VW-3, but uses a different encryption algorithm (see below).

The remaining three schemes are used in Audi vehicles from approximately 2005 until 2011 (VW-5), the VW Passat since 2005 (model B6/type 3C and newer, VW-6) and new VW vehicles like the Golf 7 (VW-7). We have not further investigated the security of these systems, but at least for older vehicles, it seems likely that similar design choices as for VW-1–VW-4 were made.

For our initial analyses, we implemented the most likely demodulation and decoding procedure for all of the above systems. We then collected rolling codes of multiple remote controls for each scheme and compared the resulting data. For all schemes VW-1–VW-4, we found that most of the packet content appeared to be encrypted, except for a fixed start pattern and the value of the pressed button sent in plain. We hence assumed that all systems use implicit authentication, i.e., check the correctness of a rolling code after decryption. Demodulation routines for VW-3 and VW-4 were independently

published in 2015 [6] after we had carried our preliminary analysis. Note that this does not cover any of the cryptographic algorithms presented here.

3.1 Analysis of Remote Control and ECU

We obtained various VW Group remote controls and extracted the Printed Circuit Boards (PCBs) for further analysis of the hardware. A typical PCB for a VW Group RKE remote includes a Microcontroller (μ C), an RF transmitter, an antenna (integrated on the PCB) and a coin cell battery as the main components. On many remote control PCBs (e.g., implementing VW-2), we found a μ C marked with Temic/Atmel M44C890E, cf. Figure 4. According to the datasheet available online [3], this μ C is a 4-bit processor, the so-called MARC4. The μ C is mask-programmed, i.e., the program code is placed in Read Only Memory (ROM) and hence fixed at manufacturing. According to Laurie [21], it is possible to re-construct the program code of MARC4 processors by taking microscopic photographs of the ROM memory and applying further image processing to extract the value of each individual bit. However, we did not follow this approach because we did not have access to suitable microscopic equipment.

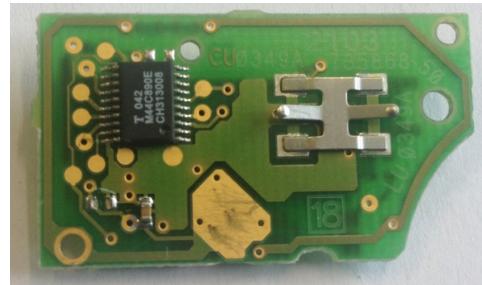


Figure 4: PCB of an older VW Group remote control using a MARC4 μ C

When studying remote controls of newer vehicles, we found different, not easily identifiable μ Cs on the PCB. An example of this is shown in Figure 5: We could not identify the type of μ C from the markings on the main IC (top, towards the right), which complicates the reverse engineering.

It seemed conceivable that some form of key derivation could be present, which would have to be implemented on the receiving ECU's side. Thus, we opted to analyze the RKE ECUs in the vehicle that receive and process the remote control signals. We therefore bought a number of ECUs implementing the respective RKE functionality, and attemp-

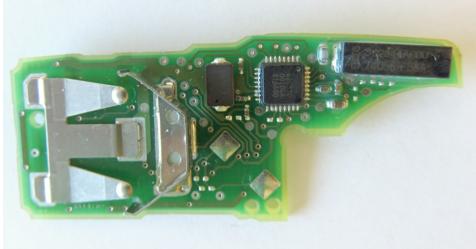


Figure 5: PCB of a newer VW Group remote control using an unidentified μ C

ted to extract the firmware of the μ Cs present on the PCB of the ECU. Note that in contrast to the low-power 4-bit or 8-bit processors usually employed in the remote control, the RKE ECUs often handles numerous additional features of the vehicle and thus utilizes a more powerful, Flash-programmable 16-bit or 32-bit μ C (with documented debug and programming interfaces).

Using widely available, standard programming tools for automotive processors, we were able to obtain firmware dumps for all studied ECUs. We then located and recovered the cryptographic algorithms by performing static analysis of the firmware image, searching amongst others for constants used in common symmetric ciphers and common patterns of such ciphers (e.g., table lookups, sequences of bitwise operations). The results of this process are described in more detail for each scheme VW-1–VW-4 in the following. Note that as part of our negotiations with VW Group, and to protect VW Group customers, we agreed to not fully disclose the part numbers of the analyzed ECUs and the employed μ Cs at this point. We furthermore agreed to omit certain details of the reverse-engineering process, as well as the values of cryptographic keys.

3.2 The VW-1 Scheme

The VW-1 system is the only VW Group scheme discussed in this paper that operates at 433.92 MHz (all newer systems use a frequency of 434.4 MHz). In contrast to newer RKE schemes, the start of a packet is not indicated by a long preamble, but by a single 1-0 pattern (500 μ s high level, 500 μ s low level). After this, the data bits are transmitted LSB-first in pulse-width encoded form: A zero is indicated by a short high level followed by a longer low level, while a one is represented with the opposite pattern (long high, short low). We discovered that the first four bytes hold the UID of the remote in an obfuscated form (several bytes of the packet are XORed). The following three bytes *lfsr* hold the

(byte-permuted) state of a Linear Feedback Shift Register (LFSR) that is clocked a fixed number of ticks for each new rolling code (i.e., the LFSR state has the role of a counter). For reasons of responsible disclosure, we do not provide the full details of the obfuscation function and the LFSR feedback in this paper. One bit of the final nibble *btn* indicates the pressed button. The overall structure of a VW-1 rolling code packet is shown in Figure 6:

UID	lfsr	btn
0	32	56 59

Figure 6: Packet structure of a rolling code for VW-1. Gray background indicates that the part is obfuscated or holds the LFSR state. The start pulse is not shown.

In conclusion, the security of the VW-1 is *solely based on obscurity*. Neither is there a cryptographic key involved in the computation of the rolling code, nor are there any vehicle or remote control specific elements for some form of key diversification. With the knowledge of the details of the obfuscation function and the LFSR, an adversary can generate valid rolling codes to open and close a VW-1 vehicle based on a single eavesdropped signal (to obtain the UID and the current state of the LFSR). Note that we observed similarly insecure LFSR-based schemes in older Audi vehicles built before 2004.

3.3 The VW-2 and VW-3 Schemes

Starting with VW-2, a rolling code packet has the following structure: A preamble (regular 0-1 pattern) is followed by a fixed start sequence *start* (individual per scheme), an encrypted 8-byte payload, and finally a byte *btn* indicating the button that was pressed. The packet structure (not showing the preamble) is depicted in Figure 7.

start	UID	ctr	btn'	btn
0	24	56	80	88 95

Figure 7: Packet structure of a rolling code for VW-2–4. Gray background indicates that the part is encrypted. Note that the fixed start pattern is shorter for VW-2.

The 8-byte payload is generated from the following plaintext: a 4-byte *UID*, a 3-byte counter *ctr*, and one byte *btn'* again indicating the pressed button. This payload is then encrypted using a proprietary block cipher that we recovered from the ECU

firmware as described in Section 3.1. We later found that this cipher appears to be the so-called AUT64 cipher employed in certain immobilizer transponders as well [4]. Hence, we will use the name AUT64 in the following and follow the notation given in the public datasheet.

AUT64 is an iterated cipher, operating on 8-byte blocks. It uses a round structure as depicted in Figure 8: In each round i the state (represented as bytes $a_0 \dots a_7$) is first byte-permuted, using a *key-dependent* permutation σ . This permutation is fully described by a $3 \cdot 2^3 = 24$ bit string. Then, bytes $a_0 \dots a_6$ are left unchanged, while byte a_7 is updated using the round function $g(a_0, \dots, a_7, key_i)$, where key_i is a 32-bit round key. In the case of AUT64 in the VW Group system, the cipher has 12 rounds, while the datasheet [4] only specifies a possible number of rounds between 8 and 24. The

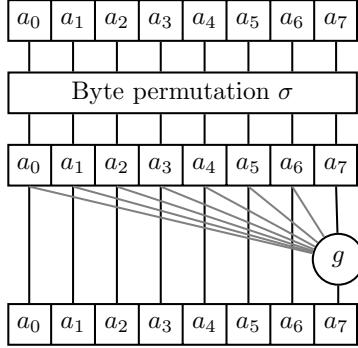


Figure 8: One round i of the AUT64 block cipher as used in VW-2 and VW-3. a_0, \dots, a_7 is the 8-byte state of the cipher, $g(a_0, \dots, a_7, key_i)$ the round function.

internal structure of g is shown in Figure 9: The input bytes a_0, \dots, a_7 are first combined with the 32-bit round key key_i using a sequence of concatenations, table look-ups, and XOR operations denoted as f . Note that the round key is derived from a part (denoted as k_f in the following) of the main key k by a fixed, nibble-wise permutation per round. Each nibble of the 8-bit output of f is then passed through the same 4-to-4 S-Box τ , bit-permuted using the same permutation σ used for the state (but applied on a bit-level), and again passed through a second instance of τ . Note that both σ and τ are *key-dependent* in addition to key_i . Hence, the full key of the AUT64 cipher is the tuple $k = (k_f, \sigma, \tau)$ with an overall key size of $32 + 3 \cdot 2^3 + 4 \cdot 2^4 = 120$ bit.

However, not all choices for τ and σ are permissible in order to have a bijective S-Box and a valid permutation—in total, there are $16!$ bijective

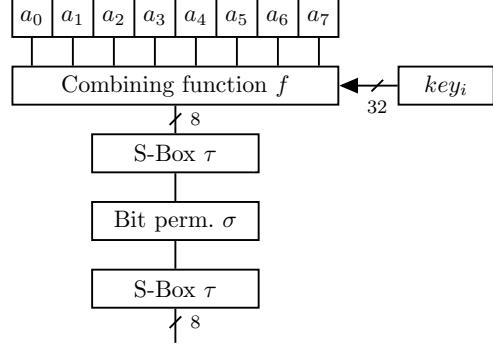


Figure 9: One round function g of the AUT64 block cipher as used in VW-2 and VW-3. a_0, \dots, a_7 is the 8-byte state of the cipher, key_i the round key.

4-to-4 S-Boxes and $8!$ permutations. This results in an effective key size of $32 + \log_2(8!) + \log_2(16!) = 91.55$ bit. Finding an AUT64 key by exhaustive search is therefore beyond current computational capabilities, where a security level of 80 bit is usually deemed acceptable for lightweight ciphers.

We have not further analyzed the mathematical security of the cipher, but believe this to be an interesting research problem, especially due to the unconventional design with several key-dependent operations. For the analysis of the VW-2 and VW-3 RKE systems, however, it turned out that no further cryptanalysis is necessary: Both schemes use a fixed, *global master key* independent of vehicle or remote control. In other words, this means that the same AUT64 key is stored in millions of ECUs and RKE remotes, *without any key diversification* being employed at all. The sole means by which the vehicle determines if a rolling code is valid is hence by white-listing certain UIDs and checking if the counter is within the validity window. Incidentally, this also implies that a VW Group vehicle using a particular scheme receives and decrypts all rolling codes for that scheme transmitted in the vicinity.

Note that the global AUT64 master keys for VW-2 and VW-3 are different, but both can be extracted from the ECU firmware and possibly from the μC in the remote control as well (e.g. with invasive attacks like micro-probing or side-channel analysis).

3.4 The VW-4 Scheme

In newer VW Group vehicles from approximately 2009 onwards, we found an RKE system that has the same encoding and packet structure as VW-3 (although with a different start pattern), but does not employ the AUT64 cipher. For this system VW-4,

the analysis of the respective ECU firmware revealed that the XTEA cipher [24] is used to encrypt a rolling code packet with a format otherwise identical to VW-3 (cf. Figure 7).

XTEA is a block cipher based on a 64-round Feistel structure with 64-bit block size and 128-bit key. Due to the structure of the round function based on Addition, Rotate, XOR (ARX) operations, it is well suited for lightweight software implementations required for low-end and low-power devices like RKE remotes. The best known cryptanalytical attack on XTEA [22] is of theoretical nature (related-key rectangle attack on 36 rounds with $2^{63.83}$ byte of data and $2^{104.33}$ steps) and hence not relevant in the context of RKE systems.

However, again we found that a *single, worldwide key* is used for all vehicles employing the VW-4 system. The same single point of failure of the older systems VW-1–VW-3 is hence also present in recently manufactured vehicles. For example, we found this scheme implemented in an Audi Q3, model year 2016, and could decrypt and generate new valid rolling codes to open and close this vehicle (and numerous other VW Group vehicles, cf. Section 3.5.1).

3.5 Implications and Observations

As the main result of this section, we discovered that the RKE systems of the majority of VW Group vehicles have been secured with only a few cryptographic keys that have been used worldwide over a period of almost 20 years. With the knowledge of these keys, an adversary only has to eavesdrop a single signal from a target remote control. Afterwards, he can decrypt this signal, obtain the current UID and counter value, and create a clone of the original remote control to lock or unlock any door of the target vehicle an arbitrary number of times.

We observed that (mostly) VW-4 vehicles blocked the original remote control if a valid rolling code with a counter more than 2 behind is received. In other words, if ctr is the value expected by the vehicle, any rolling code with $ctr - 2$ or less leads to the blocking. If an adversary sends at least two valid signals with increased counter values (e.g., “unlock” and “lock”), the original remote control of the owner will stop working in the moment when the car receives an outdated signal. In this case, usually automatic re-synchronization procedures described in the respective vehicle’s manual help technically experienced car owners to re-synchronize the remote control to the car. In contrast, if the adversary only sends a single valid signal, the original remote will not be blocked, but only operate on the second button press, be-

cause the counter in vehicle and remote are in sync afterwards. Note that the blocking behaviour could be used for an automatized Denial-of-Service (DoS) attack (aiming to lock out the legitimate car owners of affected vehicles) by intentionally sending an old signal (with a counter value of $ctr - 2$ or less).

In conclusion, while the cryptographic algorithms have improved over the years (from LFSR over AUT64 to XTEA), the crucial problem of key distribution has not been properly solved in the studied schemes VW-1–4. However, according to VW Group, this problem has been addressed in the latest generation of vehicles, where individual cryptographic keys are used. We discuss the consequences and general implications of a successful attack on a RKE system in more detail in Section 5.

3.5.1 Vulnerable Vehicles

Our findings affect amongst others the following VW Group vehicles manufactured between 1995 and 2016. Cars that we have practically tested are highlighted in bold. Note that this list is not exhaustive, as we did not have access to all types and model years of cars, and that it is unfortunately not clear if and when a car model has been upgraded to a newer scheme.

Audi: **A1**, **Q3**, R8, S3, TT, various other types of Audi cars (e.g. remote control part number 4D0 837 231)

VW: **Amarok**, (New) Beetle, Bora, **Caddy**, Crafter, e-Up, Eos, Fox, **Golf 4**, Golf 5, **Golf 6**, Golf Plus, Jetta, Lupo, **Passat**, **Polo**, **T4**, **T5**, Scirocco, **Sharan**, **Tiguan**, Touran, Up

Seat: **Alhambra**, Altea, Arosa, Cordoba, **Ibiza**, Leon, MII, Toledo

Škoda: City Go, Roomster, **Fabia 1**, **Fabia 2**, Octavia, SuperB, Yeti

It is conceivable that all VW Group (except for some Audi) cars manufactured in the past and partially today rely on a “constant-key” scheme and are thus vulnerable to the attacks described in this paper, except for those cars that rely on the latest platform, e.g., the Golf 7 for VW.

Note that identical VW Group cars are sold under different names in other countries, e.g., some Golf versions were sold as “Rabbit” in North America. We have tested some remote controls operating at 315 MHz, e.g., for the US market, and found them to be vulnerable to our attacks as well, i.e., the only difference to their European counterparts is the operating frequency. Furthermore, cars of different brands

may share the same basic technology, e.g., we found some model years of Ford Galaxy that have the same flawed RKE system as their VW Group derivatives VW Sharan and Seat Alhambra.

3.5.2 Temporary Countermeasures

Completely solving the described security problems would require a firmware update or exchange of both the respective ECU and (worse) the vehicle key containing the remote control. Due to the strict testing and certification requirements in the automotive industry and the high cost of replacing or upgrading all affected car keys in the field, it is unlikely that VW Group can roll out such an update in the short term. Hence, we give recommendations for users of affected vehicles in the following.

The well-known advice (see e.g. [25]) to verify that a vehicle was properly locked with the remote control (blinking direction lights, sound) is no longer sufficient. An adversary may have eavesdropped the “lock” signal from a distance of up to 100 m and generate a new, valid “unlock” rolling code any time later. Preventing or detecting the eavesdropping of RF signals is impractical. Hence, the only remaining (yet impractical) countermeasure is to fully deactivate or at least not use the RKE functionality and resort to the mechanical lock of the vehicle. Note that in addition, for many cars, the alarm will trigger after a while if the car doors or the trunk are mechanically opened, unless the immobilizer is disarmed with the original key.

With respect to forensics, there are several potential indicators (due to the nature of rolling code schemes) that the remote control may have been cloned: If the vehicle does not unlock on the first button press, this could imply that an adversary has sent valid rolling codes with counter values greater than the one stored in the original remote control. Note that no traces of the attack are left once the counter in the original remote control has caught up with the increased value stored in the car. Further, a complete blocking of the remote control (see above) may be an indicator (e.g., for insurance-related court cases) that the RKE system was attacked. It should however be taken into account that, according to our practical tests, the remote control will also be blocked if the car receives a counter that is increased by more than 250 compared to the last stored value—this could for example happen if the remote control buttons are pushed many times while not in the range of the vehicle.

4 Case Study 2: The Hitag2 System

The Hitag2 rolling code system is an example of a RKE scheme that is not specific to a single vehicle brand. Instead, it is implemented on the PCF7946 and PCF7947 ICs manufactured by NXP. While these ICs contain an 8-bit general-purpose μ C that (in theory) allows to realize a fully proprietary scheme [26], it appears that numerous vehicle manufacturers have used a similar (though not identical) RKE system, potentially following NXP’s reference implementation. In contrast to the VW Group system described in Section 3, it seems that manufacturers did not use a fixed, global cryptographic key. Hence, to break this system, we developed a novel attack to exploit the cryptographic weaknesses of Hitag2 in the RKE context.

We first describe the Hitag2 cipher, which was previously published in [35]. We have fully reverse-engineered the rolling code scheme used in the Hitag2 remote control ICs PCF7946/7947 as further described in Section 4.2. The analysis was done in a black-box fashion—we used a remote control for which we were able to obtain the Hitag2 key (since it was shared with the immobilizer in this particular case), guessed potential implementations (based on the immobilizer protocol) for the rolling code system, and finally recovered the complete scheme. In contrast to the analysis of the VW Group systems, no firmware extraction and reverse-engineering of program code was necessary.

To this date, the best known practical cryptanalysis of Hitag2 was proposed in [32] in the context of vehicle immobilizers. Their attack requires 136 authentication attempts and 2^{35} encryptions/lookups, which take 5 minutes on a laptop. In the context of RKE systems, gathering 136 rolling code traces is not practical in a realistic scenario, as it requires to wait for the victim to push a button on the remote that many times. We therefore propose a new attack that requires eavesdropping less authentication attempts (usually between 4 and 8) and one minute computation on a laptop. In Section 4.4, we present our novel correlation attack on Hitag2 in a RKE scenario.

We first need to introduce some notation. Let $\mathbb{F}_2 = \{0, 1\}$ the field of two elements (or the set of Booleans). The symbol \oplus denotes exclusive-or (XOR) and 0^n denotes a bitstring of n zero-bits. Given two bitstrings x and y , xy denotes their concatenation. \bar{x} denotes the bitwise complement of x . We write y_i to denote the i -th bit of y . For example, given the bitstring $y = 0x03$, $y_0 = y_1 = 0$ and $y_6 = y_7 = 1$. We denote encryptions by $\{-\}$.

4.1 Hitag2 Cipher

The targeted RKE protocol uses the Hitag2 stream cipher. This cipher has been reverse engineered in [35]. The cipher consists of a 48-bit LFSR and a non-linear filter function f . Each clock cycle, twenty bits of the LFSR are put through the filter function, generating one bit of keystream. Then the LFSR is shifted one bit to the left, using the feedback polynomial to generate a new bit on the right. See Figure 10 for a schematic representation.

Definition 4.1 *The feedback function $L: \mathbb{F}_2^{48} \rightarrow \mathbb{F}_2$ is defined by $L(x_0 \dots x_{47}) := x_0 \oplus x_2 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_8 \oplus x_{16} \oplus x_{22} \oplus x_{23} \oplus x_{26} \oplus x_{30} \oplus x_{41} \oplus x_{42} \oplus x_{43} \oplus x_{46} \oplus x_{47}$.*

The filter function f consists of three different circuits f_a, f_b and f_c , which output one bit each. The circuits f_a and f_b are employed more than once, using a total of twenty input bits from the LFSR. Their resulting bits are used as input for f_c . The circuits are represented by three Boolean tables that contain the resulting bit for each input.

Definition 4.2 (Filter function) *The filter function $f: \mathbb{F}_2^{48} \rightarrow \mathbb{F}_2$ is defined by*

$$f(x_0 \dots x_{47}) = f_c(f_a(x_2 x_3 x_5 x_6), f_b(x_8 x_{12} x_{14} x_{15}), \\ f_b(x_{17} x_{21} x_{23} x_{26}), f_b(x_{28} x_{29} x_{31} x_{33}), \\ f_a(x_{34} x_{43} x_{44} x_{46})),$$

where $f_a, f_b: \mathbb{F}_2^4 \rightarrow \mathbb{F}_2$ and $f_c: \mathbb{F}_2^5 \rightarrow \mathbb{F}_2$ are

$$f_a(i) = (0xA63C)_i \\ f_b(i) = (0xA770)_i \\ f_c(i) = (0xD949CBB0)_i.$$

Because $f(x_0 \dots x_{47})$ only depends on $x_2, x_3, x_5 \dots x_{46}$ we shall define $f_{20}: \mathbb{F}_2^{20} \rightarrow \mathbb{F}_2$, writing $f(x_0 \dots x_{47})$ as $f_{20}(x_2, x_3, x_5 \dots x_{46})$.

Remark 4.3 (Cipher schematic) *Figure 10 is different from the schematic that was introduced by [35] and later used by [11, 28, 34]. The input bits of the filter function in Figure 10 are shifted by one with respect to those of [35]. The filter function in the old schematic represents a keystream bit at the previous state $f(x_{i-1} \dots x_{i+46})$, while the one in Figure 10 represents a keystream bit of the current state $f(x_i \dots x_{i+47})$. Furthermore, we have adapted the Boolean tables to be consistent with our notation.*

4.2 Rolling Code Scheme

This section describes the rolling code scheme used by remotes based on the chips PCF7946/7947. When a button on the remote control is pressed, it

transmits a message of the form shown in Figure 11. UID is a 32-bit identifier; btn is a 4-bit button identifier; $lctr$ are the 10 least-significant bits of a 28-bit counter ctr ; ks are 32-bits of keystream; and chk is an 8-bit checksum. The checksum is computed by simply XORing each byte, i.e., computing a parity byte.

During the authentication protocol, the internal state of the stream cipher is initialized. The initial state consists of the 32-bits UID concatenated with the first 16 bits of the key k . Next, the counter ctr is incremented and then $iv = ctr \parallel btn$ is XORed with the last 32 bits of the key and shifted into the LFSR. From this point, the next 32 bits of keystream, which are output by the cipher ks , are sent as proof of knowledge of the secret key k .

4.3 Cipher Initialization

The following precisely defines the initialization of the cipher and the generation of the LFSR stream $a_0 a_1 \dots$ and the keystream ks .

Definition 4.4 *Given a key $k = k_0 \dots k_{47} \in \mathbb{F}_2^{48}$, an identifier $id = id_0 \dots id_{31} \in \mathbb{F}_2^{32}$, a counter $ctr = ctr_0 \dots ctr_{27} \in \mathbb{F}_2^{28}$, a button identifier $btn_0 \dots btn_{31} \in \mathbb{F}_2^4$ and keystream $ks = ks_0 \dots ks_{31} \in \mathbb{F}_2^{32}$, we let the initialization vector $iv \in \mathbb{F}_2^{32}$ be defined as*

$$iv = ctr \parallel btn.$$

Furthermore, the internal state of the cipher at time i is $\alpha_i := a_i \dots a_{47+i} \in \mathbb{F}_2^{48}$. Here the $a_i \in \mathbb{F}_2$ are given by

$$a_i := id_i \quad \forall i \in [0, 31] \quad (1)$$

$$a_{32+i} := k_i \quad \forall i \in [0, 15] \quad (2)$$

$$a_{48+i} := k_{16+i} \oplus iv_i \oplus f(a_i \dots a_{i+47}) \quad \forall i \in [0, 31] \quad (3)$$

$$a_{80+i} := L(a_{32+i} \dots a_{79+i}) \quad \forall i \in \mathbb{N}. \quad (4)$$

Furthermore, we define the keystream bit $ks_i \in \mathbb{F}_2$ by

$$ks_i := f(a_{32+i} \dots a_{79+i}) \quad \forall i \in [0, 31]. \quad (5)$$

Note that the a_i , α_i , and ks_i are formally functions of k , id , and iv . Instead of making this explicit by writing, e.g., $a_i(k, id, iv)$, we just write a_i where k , id , and iv are clear from the context.

4.4 A Fast Correlation Attack on Hitag2

This section describes a practical key-recovery correlation attack against Hitag2. This attack requires a minimum of four rolling codes (“traces”), but will be faster and have higher success probability if more are provided. The rolling codes can have an arbitrary counter value, i.e., do not have to be consecutive. In fact, the probability of success is higher

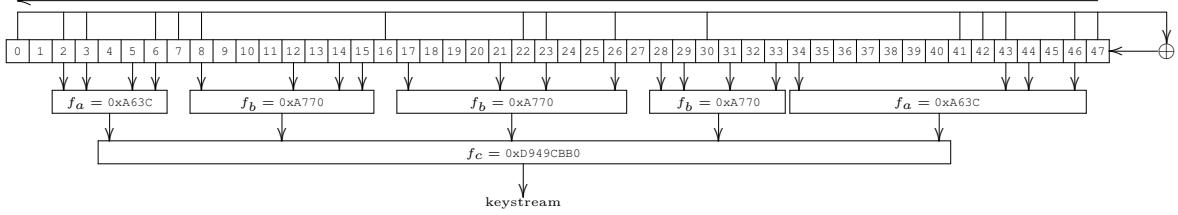


Figure 10: Structure of the Hitag2 stream cipher, based on [35]

0x0001	UID	btn	lctr	ks	0	chk
0	16	48	52	62	94 95	102

Figure 11: Packet structure of a rolling code for Hitag2. Gray background indicates the keystream part produced by the cipher.

when the traces are not consecutive, as consecutive traces often only differ in a few bits from each other, thus providing less correlation information. It also depends on whether the same button was pressed or not. The lower limit of four traces for the key recovery to work was determined experimentally. The number of consecutive traces needed is higher, usually eight. Let $\langle UID, iv^j, ks^j \rangle$, $j = 0 \dots n - 1$ be n authentication traces for $n > 3$. Then, the attacker proceeds as follows:

1. The adversary first guesses a 16-bit window corresponding to LFSR stream bits $a_{32} \dots a_{47}$. Observe that $a_{32} \dots a_{47} = k_0 \dots k_{15}$ and together with the UID, this gives the adversary LFSR bits $a_0 \dots a_{47}$, see Definition 4.4. Also note that $a_0 \dots a_{47}$ is constant over all traces. The adversary can now compute $b_0 = f(a_0 \dots a_{47})$.
2. The adversary will then shift this 16-bit window to the left of the LFSR, until bits $a_{32} \dots a_{47}$ are on the very left of the LFSR. This is the point when the cipher starts outputting ks , see Equation 5.
3. Next, the adversary will compute a correlation score for this guess. The window determines 8 input bits $x_0 \dots x_7$ to the filter function f_{20} (see Figure 10) while the remaining 12 inputs remain unknown. This correlation is taken as the ratio of those 2^{12} input values $x_8 \dots x_{19}$ that produce the correct keystream bit (ks_0). Furthermore, shifting our window further to the left allows the adversary to perform tests on multiple keystream bits ($ks_0 \dots ks_{15}$). Although, with every bit shift, the window becomes smaller as the leftmost bits will fall outside the LFSR, meaning that more input bits are unknown.

Definition 4.5 We define the single-bit correlation score as:

$$bit_score(x_0 \dots x_{n-1}, b) = \frac{\#(b = f_{20}(y_0 \dots y_{19}))}{2^{19-n}}$$

where $y_0 \dots y_{n-1} = x_0 \dots x_{n-1}$, $n < 20$ (at the first iteration of Step 3, $n=8$). We define the multiple-bit correlation score as:

$$score(x_0, ks_0) = bit_score(x_0, ks_0)$$

$$score(x_0 \dots x_{n-1}, ks_0 \dots ks_{n-1}) =$$

$$bit_score(x_0 \dots x_{n-1}, ks_{n-1}) * score(x_0 \dots x_{n-2}, ks_0 \dots ks_{n-2})$$

for $n < 20$.

The adversary will assign this guess the average score over all traces. Note that, so far this scoring computation is independent of the value iv as it happens before iv gets to have any influence on it (i.e. it is only XORed with unknown bits).

4. The adversary will now sort all guesses according to their score and store them in a table of fixed size, discarding the guesses with lowest scores when needed. Experiments show that a table of size 400,000 guesses is usually sufficient.
5. For each guess in the table, the adversary goes back to Step (1) and proceeds as before, except that she will now extend the window size by one (to size 17, ..., 32), guessing the next LFSR stream bit (a_{48}, \dots, a_{51}). The bigger window allows the adversary to test on an additional bit of keystream, giving her more meaningful correlation information each time. Special care needs to be taken at Step (3) while scoring multiple traces, since $a_{48} = k_{16+i} \oplus iv_i \oplus b_0$ (see Eq. 3) and the iv will be different in each trace. This is not a problem since in the previous Step (1) we had computed the corresponding keystream bit b_i , and iv_i is sent in clear. Therefore key bits k_{16+i} can be computed for $i \in [0, 31]$.

The power of this attack comes from using the window on the right of the LFSR to compute the necessary keystream bits to correct the internal state, while combining different traces and using the window on the left of the LFSR to get meaningful correlation information on multiple keystream bits.

4.5 Practical Results and Implications

We implemented the above correlation attack on a standard laptop. When executing this attack in practice, the first obstacle that an adversary faces is the fact that only the 10 Least Significant Bits (LSBs) of the counter ctr are sent over the air (see Figure 11), but the full 28-bit counter is used to initialize the cipher (both car and remote store the full counter). Therefore, the adversary needs to guess the remaining 18 bits. In practice, this is not a problem as it takes $2^{10} = 1024$ key pushes on the remote to have a carry to the Most Significant Bits (MSBs) and therefore this usually happens only a couple of times a year. In the worst case, the adversary has to repeat the above attack with increasing MSBs until she has the correct guess.

On average, our attack implementation recovers the cryptographic key in approximately 1 minute computation, requiring a few eavesdropped rolling codes (between 4 and 8). As mentioned, the adversary needs to repeat this computation for each guess of the 18 MSBs of the counter. For the vehicles we tested, the MSBs of the counter were usually between 0 and 10, which results in a total attack time of less than 10 min. Besides, there was a strong correlation between the vehicle’s age and the counter value, so educated guesses are also possible.

We verified our findings in practice by building a key emulator and then unlocking and locking the vehicles with newly generated rolling codes:

Manufacturer	Model	Year
Alfa Romeo	Giulietta	2010
Chevrolet	Cruze Hatchback	2012
Citroen	Nemo	2009
Dacia	Logan II	2012
Fiat	Punto	2016
Ford	Ka	2009, 2016
Lancia	Delta	2009
Mitsubishi	Colt	2004
Nissan	Micra	2006
Opel	Vectra	2008
Opel	Combo	2016
Peugeot	207	2010
Peugeot	Boxer	2016
Renault	Clio	2011
Renault	Master	2011

The vehicles in the above list are our own and also from colleagues and friends who volunteered. We furthermore found the following list of supported vehicles for an after-market universal remote control [19] that is presumably implementing the

Hitag2 RKE scheme: *Abarth 500, Punto Evo; Alfa Romeo Giulietta, Mito; Citroen Jumper, Nemo; Fiat 500, Bravo, Doblo, Ducato, Fiorino, Grande Punto, Panda, Punto Evo, Qubo; Dacia Duster; Ford Ka; Lancia Delta, Musa; Nissan Pathfinder, Navara, Note, Qashqai, X-Trail; Opel Corsa, Meriva, Zafira, Astra; Peugeot Boxer, Expert; and Renault Clio, Modus, Trafic, Twingo*. This list includes most of our tested vehicles. This would indicate that all vehicles mentioned in the list (although not practically tested by us) are vulnerable to the described attacks as well.

In contrast to the VW Group scheme, the vulnerabilities in the Hitag2 RKE system are caused by the cryptographically weak cipher, not a weak key distribution method. In consequence, even though it must be said that the correlation attack of Section 4.4 is devastating from a cryptographic point of view, the data complexity is slightly higher compared to the VW Group schemes, which can be broken with one single eavesdropped signal. The attack on Hitag2 requires at least four (not necessarily consecutive) rolling codes, i.e., the adversary has to be present for a longer period of time to capture signals for multiple key presses on the victim’s remote control.

However, to quickly obtain the required rolling codes, the adversary could selectively jam the signal during the final checksum byte (which is predictable). In this case, the vehicle ignores the rolling code, but the adversary nevertheless obtains the keystream. The victim would hence notice that the vehicle does not respond, and instinctively press the button repeatedly. After having received the fourth signal, the adversary stops jamming and the remote control operates normally from the victim’s point of view. However, the attacker has then collected the required amount of rolling codes to subsequently extract the cryptographic key. Hence, if the described behaviour is observed by a vehicle owner, it is an indication that an attack may be in progress.

5 Conclusion

Answering the original research question about the security of automotive RKE systems, the results of this paper show that major manufacturers have used insecure schemes over more than 20 years. Due to the widespread use of the analyzed systems, our findings have worldwide impact. Owners of affected vehicles should be aware that unlocking the doors of their car is much simpler than commonly assumed today. Both for the VW Group and the Hitag2 rolling code schemes, it is possible to clone the original remote control and gain unauthorized access to

the vehicle after eavesdropping one or a few rolling codes, respectively. The necessary equipment to receive and send rolling codes, for example SDRs like the USRP or HackRF and off-the-shelf RF modules like the TI Chronos smart watch, are widely available at low cost. The attacks are hence highly scalable and could be potentially carried out by an unskilled adversary. Since they are executed solely via the wireless interface, with at least the range of the original remote control (i.e., a few tens of meters), and leave no physical traces, they pose a severe threat in practice.

Security and Safety Implications The implications of our findings are manifold: Personal belongings left in a locked vehicle (as well as vehicle components like the infotainment system) could be stolen if a thief uses the vulnerabilities of the RKE system to unlock the vehicle after the owner has left. This approach is considerably more stealthy and harder to prevent than the currently known methods of theft (e.g., using physical force or jamming the rolling code). Moreover, since a valid rolling code usually disables the alarm system, the theft is more likely to remain undetected for a longer period of time. Common recommendations like “lock it or lose it” [25] or “verify that the vehicle has been successfully locked and the transmission has not been jammed” (blinking direction lights, sound) are hence no longer sufficient to effectively prevent theft. A successful attack on the RKE and anti-theft system would also enable or facilitate other crimes:

- theft of the vehicle itself by circumventing the immobilizer system (e.g. [32, 33]) or by programming a new key into the car via the OBD port with a suitable tool
- compromising the board computer of a modern vehicle [10, 20], which may even affect personal safety, e.g., by deactivating the brakes while switching on the wiping system in a bend
- inconspicuously placing an object or a person inside the car. The car could be locked again after the act
- on-the-road robbery, affecting the personal safety of the driver or passengers if they (incorrectly) assume that the vehicle is securely locked

Note that due to the long range of RKE systems it is technically feasible to eavesdrop the signals of all cars on a parking lot or at a car dealer by placing an eavesdropping device there overnight. Afterwards, all vulnerable cars could be opened by the adversary. Practical experiments suggest that the

receiving ranges can be substantially increased: The authors of [18] report eavesdropping of a 433 MHz RFID system, with technology comparable to RKE, from up to 1 km using low-cost equipment. Likewise, a large-scale DoS attack targeting VW Group cars would be possible with an automated approach—as a result, the RKE system of the vulnerable vehicle types would be deactivated for the respective remote control and VW Group would face increased demand for customer service, i.e., re-synchronizing remotes.

Legal Implications, Forensics, and Counter-measures It is unclear whether such attacks on the RKE scheme are currently carried out in the wild by criminals. However, there have been various media reports about unexplained theft from locked vehicles in the last years. The security issues described in this paper could explain such incidents. Note that we have analyzed further automotive RKE systems (with similar results regarding their (in)security), but due to the difficulty of responsible disclosure, cannot publish all results at this point.

As of today, even experts in car theft cases expressed the opinion that the alarm and electronic door locking systems of a car cannot be easily circumvented. From now on, they have to consider that special universal remote controls to bypass the security mechanisms might be used by criminals. In contrast to mechanical tools to open vehicles, such a device would leave no physical traces. Insurance companies may thus have to accept that certain car theft scenarios that have so far been regarded as insurance fraud (e.g. theft of personal belongings out of a locked car without physical traces) have, considering the results of this paper, a higher probability to be real. From a forensics point of view, the need to press the button of the remote control more than once in order to unlock the vehicle is an indicator that the car might have been accessed by a criminal. For VW Group vehicles, the “blocking” of a remote control should be regarded as suspicious as well. However, there are other causes for such behaviour, e.g., short range due to an empty battery of the remote control or environmental RF noise.

While the vulnerabilities of the VW Group system are due to worldwide master keys, Hitag2-based systems suffer from weaknesses in the cipher itself. Hence, in conclusion, for a “good” RKE system, both secure cryptographic algorithms (e.g., AES) and secure key distribution are necessary. Techniques to solve the security problems discovered in this paper are widely available [23]. Atmel has created an open RKE protocol design [5], which is pub-

lished in full detail. The security of their design was scrutinized by Tillich et al. in [29]. It is now up to vehicle manufacturers to securely implement such next-generation RKE schemes.

For owners of affected vehicles, as a temporary countermeasure in cases where valuable items are left in the vehicle, we can unfortunately only recommend to stop using or disable/remove the RKE part of the car key and fall back to the mechanical lock: **Lock It or Lose It? Remove It!**

6 Responsible Disclosure

Regarding the vulnerabilities of VW Group systems, we contacted VW Group first in November 2015. We discussed our findings in a meeting with VW Group and an affected sub-contractor in February 2016, before submitting the paper. VW Group received a draft version of this paper and the final version. VW Group acknowledged the vulnerabilities. As mentioned in the paper, we agreed to leave out amongst others the following details: cryptographic keys, part numbers of vulnerable ECUs, and the used programming devices and details about the reverse-engineering process.

For Hitag2, we notified NXP in January 2016. NXP received a version of this paper before submission. We would like to mention that the fact that Hitag2 is cryptographically broken has been publicly known for several years and NXP has already informed their customers back in 2012. We would further like to highlight that for several years, NXP offers newer, AES-based RKE ICs that are not affected by the vulnerabilities described in this paper. Furthermore, many car manufacturers have already started using the more secure chips for new designs.

References

- [1] ABC7NEWS. Key fob car thefts, 2013. <http://abc7news.com/archive/9079852>.
- [2] ARSTECHNICA. After burglaries, mystery car unlocking device has police stumped, 2013. <http://arstechnica.com/security/2013/06/after-burglaries-mystery-car-unlocking-device-has-police-stumped>.
- [3] ATMEL. M44C890 Low-Current Microcontroller for Wireless Communication , 2001. datasheet, available at <http://pdf1.alldatasheet.com/datasheet-pdf/view/118247/ATMEL/M44C890.html>.
- [4] ATMEL. e5561 Standard Read/Write Crypto Identification IC, 2006. datasheet, available at <http://www.usmartcards.com/media/downloads/366/Atmel%20e5561%20pdf-190.pdf>.
- [5] ATMEL. Embedded AVR Microcontroller Including RF Transmitter and Immobilizer LF Functionality for Remote Keyless Entry - ATA5795C. datasheet, available at http://www.atmel.com/images/Atmel-9182-Car-Access-ATA5795C_Datasheet.pdf, November 2014.
- [6] BLOESSL, B. gr-keyfob. Github repository, 2015. <https://github.com/bastibl/gr-keyfob>.
- [7] BOGDANOV, A. Attacks on the KeeLoq Block Cipher and Authentication Systems. In *Workshop on RFID Security (RFID-Sec'08)* (2007). rfidsec07.etsit.uma.es/slides/papers/paper-22.pdf.
- [8] BONO, S. C., GREEN, M., STUBBLEFIELD, A., JUELS, A., RUBIN, A. D., AND SZYDLO, M. Security analysis of a cryptographically-enabled RFID device. In *14th USENIX Security Symposium (USENIX Security 2005)* (2005), USENIX Association, pp. 1–16.
- [9] CESARE, S. Breaking the security of physical devices. Presentation at Blackhat'14, August 2014.
- [10] CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., SAVAGE, S., KOSCHER, K., CZEKIS, A., ROESNER, F., AND KOHNO, T. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium (USENIX Security 2011)* (2011), USENIX Association, pp. 77–92.
- [11] COURTOIS, N. T., O'NEIL, S., AND QUISQUATER, J.-J. Practical algebraic attacks on the Hitag2 stream cipher. In *12th Information Security Conference (ISC 2009)* (2009), vol. 5735 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 167–176.
- [12] EISENBARTH, T., KASPER, T., MORADI, A., PAAR, C., SALMASIZADEH, M., AND SHALMANI, M. T. M. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme. In *Advances in Cryptology – CRYPTO'08* (2008), vol. 5157 of *LNCS*, Springer, pp. 203–220.

- [13] EUROPEAN AUTOMOBILE MANUFACTURERS ASSOCIATION. New passenger car registrations, 2015. available at http://www.acea.be/uploads/press_releases_files/20151016_PRPC_1509_FINAL.pdf.
- [14] FRANCILLON, A., DANEV, B., AND CAPKUN, S. Relay attacks on passive keyless entry and start systems in modern cars. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011* (2011), The Internet Society.
- [15] INDESTEEGE, S., KELLER, N., DUNKELMANN, O., BIHAM, E., AND PRENEEL, B. A practical attack on KeeLoq. In *27th International Conference on the Theory and Application of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT 2008)* (2008), vol. 4965 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–8.
- [16] KAMKAR, S. Drive It Like You Hacked It: New Attacks and Tools to Wirelessly Steal Cars. Presentation at DEFCON 23, August 2015.
- [17] KASPER, M., KASPER, T., MORADI, A., AND PAAR, C. Breaking KeeLoq in a Flash: On Extracting Keys at Lightning Speed. In *Progress in Cryptology - AFRICACRYPT'09* (2009), B. Preneel, Ed., vol. 5580 of *LNCS*, Springer, pp. 403–420.
- [18] KASPER, T., OSWALD, D., AND PAAR, C. Wireless security threats: Eavesdropping and detecting of active RFIDs and remote controls in the wild. In *19th International Conference on Software, Telecommunications and Computer Networks - SoftCOM'11* (2011), pp. 1–6.
- [19] KEYLINE S.P.A. RK60 guide, 2015. available at http://www.keyline.it/files/teste-elettroniche/electronic_heads_guide_13316.pdf.
- [20] KOSCHER, K., CZEKSIS, A., ROESNER, F., PATEL, F., KOHNO, T., CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., AND SAVAGE, S. Experimental security analysis of a modern automobile. In *31rd IEEE Symposium on Security and Privacy (S&P 2010)* (2010), IEEE Computer Society, pp. 447–462.
- [21] LAURIE, A. Fun with Masked ROMs — Atmel MARC4. Blog entry, 2013. <http://adamsblog.aperturelabs.com/2013/01/fun-with-masked-roms.html>.
- [22] LU, J. Related-key rectangle attack on 36 rounds of the XTEA block cipher. *International Journal of Information Security* 8, 1 (2008), 1–11.
- [23] MORADI, A., AND KASPER, T. A new remote keyless entry system resistant to power analysis attacks. In *Information, Communications and Signal Processing – ICICS 2009* (2009), IEEE, pp. 1–6.
- [24] NEEDHAM, R. M., AND WHEELER, D. J. TEA extensions. *Technical Report, Cambridge University, UK* (1997).
- [25] NEWPORT BEACH PD. Lock It Or Lose It - Newport Beach Vehicle Crime, 2011. Video available at <https://www.youtube.com/watch?v=Mmi2LRF7a18>.
- [26] PHILIPS. PCF7946AT – Security Transponder Plus Remote Keyless Entry, 1999. datasheet, available at <http://www.datasheet4u.com/pdf/PCF7946AT-pdf/609011>.
- [27] SPENCERWHYTE. Jam Intercept and Replay Attack against Rolling Code Key Fob Entry Systems using RTL-SDR. Website, retrieved January 21, 2016, March 2014. <http://spencerwhyte.blogspot.ca/2014/03/delay-attack-jam-intercept-and-replay.html>.
- [28] SUN, S., HU, L., XIE, Y., AND ZENG, X. Cube cryptanalysis of Hitag2 stream cipher. In *10th International Conference on Cryptology and Network Security (CANS 2011)* (2011), vol. 7092 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 15–25.
- [29] TILLICH, S., AND WÓJCIK, M. Security analysis of an open car immobilizer protocol stack. In *10th International Conference on Applied Cryptography and Network Security (ACNS 2012)* (2012).
- [30] VERDULT, R. *The (in)security of proprietary cryptography*. PhD thesis, Radboud University, The Netherlands and KU Leuven, Belgium, April 2015.
- [31] VERDULT, R., AND GARCIA, F. D. Cryptanalysis of the Megamos Crypto automotive immobilizer. *USENIX ;login:* 40, 6 (2015), pp. 17–22.
- [32] VERDULT, R., GARCIA, F. D., AND BALASCH, J. Gone in 360 seconds: Hijacking with Hitag2. In *USENIX Security Symposium* (August

- 2012), USENIX Association, pp. 237–252. <https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final95.pdf>.
- [33] VERDULT, R., GARCIA, F. D., AND EGE, B. Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer. In *22nd USENIX Security Symposium (USENIX Security 2013)* (2015), USENIX Association, pp. 703–718.
- [34] ŠTEMBERA, P., AND NOVOTNÝ, M. Breaking Hitag2 with reconfigurable hardware. In *14th Euromicro Conference on Digital System Design (DSD 2011)* (2011), IEEE Computer Society, pp. 558–563.
- [35] WIENER, I. Philips/NXP Hitag2 PCF7936/46/47/52 stream cipher reference implementation. <http://cryptolib.com/ciphers/hitag2/>, 2007.

OBLIVP2P: An Oblivious Peer-to-Peer Content Sharing System

Yaoqi Jia^{1*} *Tarik Moataz*^{2*} *Shruti Tople*^{1*} *Prateek Saxena*¹

¹*National University of Singapore*

{jiayaoqi, shruti90, prateeks}@comp.nus.edu.sg

²*Colorado State University and Telecom Bretagne*

tarik.moataz@colostate.edu

Abstract

Peer-to-peer (P2P) systems are predominantly used to distribute trust, increase availability and improve performance. A number of content-sharing P2P systems, for file-sharing applications (e.g., BitTorrent and Storj) and more recent peer-assisted CDNs (e.g., Akamai Netsession), are finding wide deployment. A major security concern with content-sharing P2P systems is the risk of long-term *traffic analysis* — a widely accepted challenge with few known solutions.

In this paper, we propose a new approach to protecting against persistent, global traffic analysis in P2P content-sharing systems. Our approach advocates for hiding data access patterns, making P2P systems *oblivious*. We propose OBLIVP2P— a construction for a scalable distributed ORAM protocol, usable in a real P2P setting. Our protocol achieves the following results. First, we show that our construction retains the (linear) scalability of the original P2P network w.r.t the number of peers. Second, our experiments simulating about 16,384 peers on 15 Deterlab nodes can process up to 7 requests of 512KB each per second, suggesting usability in moderately latency-sensitive applications as-is. The bottlenecks remaining are purely computational (not bandwidth). Third, our experiments confirm that in our construction, no centralized infrastructure is a bottleneck — essentially, ensuring that the network and computational overheads can be completely offloaded to the P2P network. Finally, our construction is highly parallelizable, which implies that remaining computational bottlenecks can be drastically reduced if OBLIVP2P is deployed on a network with many real machines.

1 Introduction

Content sharing peer-to-peer (P2P) systems, especially P2P file-sharing applications such as BitTorrent [1], Storj [2] and Freenet [3] are popular among users for

sharing files on the Internet. More recently, peer-assisted CDNs such as Akamai Netsession [4] and Squirrel [5] are gaining wide adoption to offload web CDN traffic to clients. The convenient access to various resources attract millions of users to join P2P networks, e.g., BitTorrent has over 150 million active users per month [6] and its file-sharing service contributes 3.35% of all worldwide bandwidth [7]. However, the majority of such P2P applications are susceptible to long-term traffic analysis through global monitoring; especially, analyzing the *pattern* of communication between a sender and a receiver to infer information about the users. For example, many copyright enforcement organizations such as IFPI, RIAA, MPAA, government agencies like NSA and ISP's are reported to globally monitor BitTorrent traffic to identify illegal actors. Monitoring of BitTorrent traffic has shown to reveal the data requested and sent by the peers in the network [8–10]. Unfortunately, while detecting copyright infringements is useful, the same global monitoring is applicable to *any* user of the P2P network, and can therefore collect benign users' data. Thus, users of such P2P systems are at a risk of leaking private information such as the resources they upload or download.

To hide their online traces, users today employ anonymous networks as a solution to conceal their digital identities or data access habits. Currently, anonymous networks include Mix networks [11–13], and Onion routing/Tor-based systems [14–17], as well as other P2P anonymity systems [18–23]. Such systems allow the user to be *anonymous*, so that the user is unidentifiable within a set of users [24].

Although above solutions provide an anonymity guarantee, they are vulnerable to long-term traffic pattern analysis attacks, which is an important threat for P2P systems like BitTorrent [25–30]. Researchers have demonstrated attacks targeting BitTorrent users on top of Tor that reveal information related to the resources uploaded or downloaded [31, 32]. Such attacks raise the question - *is anonymizing users the right defense against traffic*

*Lead authors are alphabetically ordered.

pattern analysis in P2P content sharing systems?

In this paper, we investigate a new approach to solve the problem of persistent analysis of data communication patterns. We advocate that data / resource access pattern hiding is an important and necessary step to thwart leakage of users data in P2P systems. To this end, we present a first candidate solution, OBLIVP2P—an oblivious protocol for peer-to-peer content sharing systems. Hiding data access patterns or making them *oblivious* unlinks user’s identity from her online traces, thereby defending against long-term traffic monitoring.

1.1 Approach

For hiding data access patterns between a trusted CPU and an untrusted memory, Goldreich and Ostrovsky proposed the concept of an Oblivious RAM (ORAM) [33]. We envision providing similar obliviousness guarantees in P2P systems, and therefore select ORAM as a starting point for our solution. To the best of our knowledge, OBLIVP2P is the first work that adapts ORAM to accesses in a P2P setting. However, directly employing ORAM to hide access patterns in a P2P system is challenging. We outline two key challenges in designing an oblivious and a scalable P2P protocol using ORAM.

Obliviousness. The first challenge arises due to the difference in the setting of a standard ORAM as compared to a P2P content sharing system. Classical ORAM solutions consists of a single client which securely accesses an untrusted storage (server), wherein the client is eventually the owner and the only user of the data in the memory. In contrast, P2P systems consist of a set of trusted trackers managing the network, and multiple data owners (peers) in the network. Each peer acts both as a client as well as a server in the network i.e., a peer can either request for a data or respond to other peer’s request with the data stored on its machine. Hence, adversarial peers present in the network can see the plaintext and learn the data requested by other peers, a threat that does not exists in the traditional ORAM model where only encrypted data is seen by the servers.

Scalability. The second challenge lies in seeking an oblivious P2P system that 1) the throughput scales linearly with the number of peers in the network, 2) has no centralized bottleneck and 3) can be parallelized with an overall acceptable throughput. In standard ORAM solutions, the (possibly distributed) server is responsible for serving all the data access requests from a client one-by-one. In contrast, P2P systems operate on a large-scale with multiple peers (clients) requesting resources from each other simultaneously without overloading a particular entity. To retain scalability of P2P systems, it is necessary to ensure that requests can be served by dis-

tributing the communication and computation overhead.

Solution Overview. We start with a toy construction (OBLIVP2P-0) which directly adapts ORAM to a P2P setting, and then present our main contribution which is a more efficient solution (OBLIVP2P-1).

Centralized Protocol (OBLIVP2P-0): Our centralized protocol or OBLIVP2P-0, is a direct adaptation of ORAM in a P2P system. The peers in the network behave both like distributed storage servers as well as clients. They request a centralized, trusted tracker to access a particular resource. The tracker performs all the ORAM operations to fetch the resource from the network and returns it to the requesting peer. However, this variant of OBLIVP2P protocol has limited scalability as it assigns heavy computation to the tracker, making it a bottleneck.

Distributed Protocol (OBLIVP2P-1): As our main contribution, we present OBLIVP2P-1 which provides both obliviousness and scalability properties in a tracker-based P2P system. To attain scalability, the key idea is to avoid any single entity (say the tracker) as a bottleneck. This requires distributing all the ORAM operations for fetching and sharing of resources among the peers in the network, while still maintaining obliviousness guarantees. To realize such a distributed protocol, our main building block, which we call *Oblivious Selection* (OblivSel), is a novel combination of private information retrieval with recent advances in ORAM. Oblivious Selection gives us a scalable way to securely distribute the load of the tracker. Our construction is proven secure in the honest-but-curious adversary model. Constructions and proofs for arbitrarily malicious fraction of peers is slated for future work.

1.2 System and Results

We provide a prototype implementation of both OBLIVP2P-0 and OBLIVP2P-1 protocols in Python. Our source code is available online [34]. We experimentally evaluate our implementation on DeterLab testbed with 15 servers simulating up to 2^{14} peers in the network. Our experiments demonstrate that OBLIVP2P-0 is limited in scalability with the tracker as the main bottleneck. The throughput for OBLIVP2P-1, in contrast, scales linearly with increase in the number of peers in the network. It attains an overall throughput of 3.19 MBps for a network of 2^{14} peers that corresponds to 7 requests per second for a block size of 512 KB. By design, OBLIVP2P-1 is embarrassingly parallelizable over the computational capacity available in a real P2P network. Further, our protocol exhibits no bottleneck on a single entity in experiment, thereby confirming that the network and the computational overhead can be completely offloaded to

the P2P network.

Contributions. We summarize our contributions below:

- **Problem Formulation.** We formulate the problem of making data access pattern oblivious in P2P systems. This is a necessary and important step in building defenses against long-term traffic analysis.
- **New Protocols.** We propose OBLIVP2P—a first candidate for an oblivious peer-to-peer protocol in content sharing systems. Our main building block is a primitive which we refer to as oblivious selection that makes a novel use of recent advances in Oblivious RAM combined with private information retrieval techniques.
- **System Implementation & Evaluation.** Our prototype implementation is available online [34]. We experimentally evaluate our protocol to measure the overall throughput of our system, latency for accessing resources and the impact of optimizations on the system throughput.

2 Problem

Many P2P applications are not designed with security in mind, making them vulnerable to traffic pattern analysis. We consider BitTorrent as our primary case study. However, the problem we discuss is broadly applicable to other P2P file sharing systems like Gnutella [35], Freenet [3] and Storj [2] or peer-assisted CDNs such as Akamai Netsession [4], Squirrel [5] and APAC [36].

2.1 BitTorrent: A P2P Protocol

The BitTorrent protocol allows sharing of large files between users by dividing it into blocks and distributing it among the peers. It has a dynamic network, made up of a number of nodes that join the network and volunteer themselves as peers. Each peer holds data blocks in its local storage and acts both as a client / requester and server / sender simultaneously. There exists a tracker that tracks which peers are downloading / uploading which file and saves the state of the network. It keeps information regarding the position or the IP addresses of peers holding each resource but does not store any real data blocks. A peer requests the tracker for a particular resource and the tracker responds with a set of IP addresses of peers holding the resource. The requester then communicates with these IP addresses to download the blocks of the desired resource. The peers interact with each other using a P2P protocol¹. The requester concatenates all the blocks received to construct the entire resource.

¹We want to emphasize that there are other models of P2P networks without tracker based on DHT that we are not addressing in this work.

2.2 Threat Model

In our threat model, we consider the tracker as a trusted party and peers as passive honest-but-curious adversaries i.e., the peers are expected to correctly follow the protocol without deviating from it to learn any extra information. In P2P systems including CDNs (content delivery networks) and BitTorrent, passive monitoring is already a significant threat on its own. We consider the following two types of adversaries:

Global Passive Adversary. Since BitTorrent traffic is public, there exist tools like Global BitTorrent Monitor [37] or BitStalker [38] that support accurate and efficient monitoring of BitTorrent. Previous research has shown that any BitTorrent user can be logged within a span of 3 hours, revealing his digital identity and the content downloaded [39]. Further, the adversary can log the communication history of the network traffic to perform offline analysis at a later stage. Hence, we consider it rational to assume the presence of a global adversary with the capability to observe long term traffic in the network.

Passive Colluding Peers. Some of the peers in the P2P network can be controlled by the global adversary. They can further collude to exchange data with other adversarial peers in the system. While colluding these “sybil” peers can share information such as observed / served requests and the contents stored at their local storage. Their goal is to collectively glean information about other peers in the network. A formal definition of passive colluding peers is as follows:

Definition 2.1. (*Passive Colluding peers*) We say that a peer P_i passively colludes with peer P_j if both peers share their views without any modification, where a view consists of: a transcript of the sequence of all accesses made by P_i , a partial or total copy of peer’s private storage, and a transcript of the access pattern induced by the sequence of accesses. We denote by $\mathcal{C}(P_i)$ the set of colluding peers with P_i .

Note that from the above definition, we have a symmetric relation such that if $P_i \in \mathcal{C}(P_j)$ for $i \neq j$, then $P_j \in \mathcal{C}(P_i)$. It follows that if $P_i \notin \mathcal{C}(P_j)$, then $\mathcal{C}(P_i)$ and $\mathcal{C}(P_j)$ are disjoint.

Our protocol tolerates a fraction of c adversarial peers in the network such that $c \in O(N^\varepsilon)$, where N is the total number of peers in the network and $\varepsilon < 1$. Although the P2P network undergoes churn, we assume the fraction of adversarial peers c remains within the asymptotic bounds of $O(N^\varepsilon)$. Our choice of the upper bound for c ensures an exponentially small advantage to the attacker; for an application that can tolerate higher attacker’s advantage, a larger malicious fraction can be allowed.

2.3 Insufficiency of Existing Approaches

Existing techniques propose anonymizing users to prevent traffic pattern analysis attacks. However, these solutions are not sufficient to protect against a global adversary with long term access to communication patterns.

Unlinkability Techniques (e.g. Mixnet). Existing anonymity approaches “unlink” the sender from the receiver (see survey [40]). Chaum proposed the first anonymous network called mix network [11], which shuffles messages from multiple senders using a chain of proxy servers and sends them to the receiver. Another recent system called Riposte guarantees traffic analysis resistance by unlinking a sender from its message [41]. However, all these systems are prone to attack if an adversary can observe multiple request rounds in the network.

For example, consider that *Alice* continuously communicates with *Bob* using a mixnet service. A global adversary observes this communication for a couple of rounds, and records the recipient set in each round. Let the senders’ set consists of $S_1 = \{Alice, a, b, c\}$ and $S_2 = \{a', b', Alice, c'\}$, and the recipients’ set consists of $R_1 = \{x, y, z, Bob\}$ and $R_2 = \{x', y', Bob, z'\}$ for rounds 1 and 2 respectively. The attacker can then infer the link between sender and receiver by intersecting $S_1 \cap S_2 = \{Alice\}$ and $R_1 \cap R_2 = \{Bob\}$. The attacker learns that *Alice* is communicating with *Bob*, and thus breaks the unlinkability. This attack is called the *intersection, hitting set* or *statistical disclosure attack* [25, 26]. Overall, one time unlinkability is not a sufficient level of defense when the adversary can observe traffic for arbitrary rounds.

Path Non-Correlation (e.g. Onion routing). Another approach for guaranteeing anonymity is to route the message from a path such that the sender and the receiver cannot be correlated by a subset of passive adversarial nodes. Onion-routing based systems like Tor enable anonymous communication by using a sequence of relays as intermediate nodes (called circuit) to forward traffic [15, 42]. However, Tor cannot provide sender anonymity when the attacker can see both the ends of the communication, or if a global adversary observes the entire network. Hence, if an attacker controls the entry and the exit peer then the adversarial peers can determine the recipient identity with which the initiator peer is communicating [27–30]. This is a well-known attack called the *end-to-end correlation attack* or *traffic confirmation attack* [43, 44].

2.4 Problem Statement

Our goal is to design a P2P protocol that prevents linking a user to a requested resource using traffic pattern analysis. Section 2.3 shows how previous anonymity based

solutions are susceptible to attacks in our threat model. In this work, we address this problem from a new viewpoint, by making the communication pattern oblivious in the network. We advocate that hiding data / resource access pattern is a necessary and important step in designing traffic pattern analysis resistant P2P systems.

In a P2P system such as BitTorrent, a user accesses a particular resource by either downloading (Fetch) or uploading (Upload) it to the network. We propose to build an oblivious P2P content sharing protocol (OBLIVP2P) that hides the data access patterns of users in the network. We formally define an Oblivious P2P protocol as follows:

Definition 2.2. (Oblivious P2P): Let (P_1, \dots, P_n) and \mathcal{T} be respectively a set of n peers and a tracker in a P2P system. We denote by $\vec{x}_i^j = (x_{i,1}, \dots, x_{i,M})$ a sequence of M accesses made by peer P_i such that $x_{i,j} = (op_{i,j}, fid_{i,j}, file_{i,j})$ where $op_{i,j} = \{\text{Upload}, \text{Fetch}\}$, $fid_{i,j}$ is the filename being accessed, and $file_{i,j}$ is the set of blocks being written in the network if $op_{i,j} = \text{Upload}$.

We denote by $\mathcal{A}(\vec{x}_i^j)$ the access pattern induced by the access sequence \vec{x}_i^j of peer P_i . The access pattern is composed of the memory arrays of all peers accessed while running the sequence \vec{x}_i^j . We say that a P2P is oblivious if for any two equal-length access sequences \vec{x}_i^j and \vec{x}_j^i by two peers P_i and P_j such that

- $P_j \notin \mathcal{C}(P_i)$
- $\forall k \in [M] : x_{i,k} = \text{Fetch} \Leftrightarrow x_{j,k} = \text{Fetch} \wedge x_{i,k} = \text{Upload} \Leftrightarrow x_{j,k} = \text{Upload}$
- $\forall k \in [M], |file_{i,k}| = |file_{j,k}|$

are indistinguishable for all probabilistic poly-time adversaries except for $\mathcal{C}(P_i)$, $\mathcal{C}(P_j)$, and tracker \mathcal{T} .

Scope. OBLIVP2P guarantees resistance against persistent communication traffic analysis i.e., observing the path of communication and thereby linking a sender to a particular resource. OBLIVP2P does not prevent against:

a) *Active Tampering:* An adversarial peer can tamper, alter and deviate from the protocol to learn extra information. Admittedly, this can have an impact on obliviousness, correctness and availability of the network.

b) *Side Channels:* An adversary can monitor any peer in the system to infer its usage’s habits via side channels: the number of requests, time of activity, and total number of uploads. In addition, an adversary can always infer the total file size that any peer is downloading or uploading to the P2P network. Literature shows that some attacks such as website fingerprinting can be based on the length of file requested by peers [45].

c) *Orthogonal Attacks:* Other attacks in P2P file sharing

systems consist of threats such as poisoning of files by uploading corrupted, fake or misleading content [46] or denial of service attacks [47]. However, these attacks do not focus on learning private information about the peers and hence are orthogonal to our problem.

Admittedly, our assumption about honest-but-curious is less than ideal and simplifies analysis. We hope that our construction spurs future work on tackling the active or arbitrary malicious adversaries. Emerging trusted computing primitives (e.g., Intel SGX [48]) or cryptographic measures [49] are promising directions to investigate. Lastly, OBLIVP2P should not be confused with traditional anonymous systems where a user is anonymous among a set of users. OBLIVP2P does not guarantee sender or receiver anonymity, but hides data access patterns of the users.

3 Our Approach

As a defense against traffic pattern analysis, we guarantee oblivious access patterns in P2P systems. We consider Oblivious RAM as a starting point.

3.1 Background: Tree-Based ORAM

Oblivious RAM, introduced by Goldreich and Ostrovsky [33], is a cryptographic primitive that prevents an adversary from inferring any information via the memory access pattern. Tree-based ORAM introduced by Shi et al. [50] offers a poly-logarithmic overhead which is further reduced due to improvements suggested in the follow up works [51–56]. In particular, we use Ring ORAM, [52], one of the latest improvements for tree-based ORAM in our protocol. In Ring ORAM, to store N data blocks, the memory is organized in a (roughly) $\log N$ -height full binary tree, where each node contains z real blocks and s dummy blocks. Whenever a block is accessed in the tree, it is associated to a new randomly selected leaf identifier called, tag. The client stores this association in a position map PosMap along with a private storage (stash). To read and write to the untrusted memory, the client performs an Access followed by an Evict operation described at a high level as follows:

- **Access(adr):** Given address adr , the client fetches the leaf identifier tag from PosMap. Given tag, the client downloads one block per every node in the path $\mathcal{P}(tag)$ that starts from the root and ends with the leaf tag. The client decrypts the retrieved blocks, and retrieves the desired block. This block is appended to the stash.
- **Evict(A, v):** After A accesses, the client selects a path $\mathcal{P}(v)$ based on a deterministic reverse lexicographic order, downloads the path, decrypts it and appends it to the stash. The client runs the

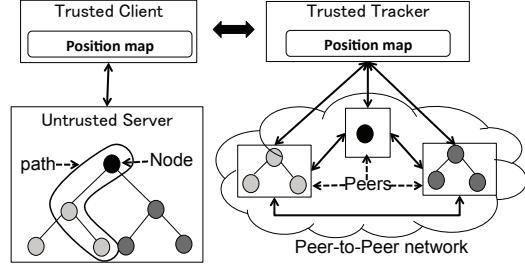


Figure 1: Mapping of a client / server ORAM model to a P2P system

least common ancestor algorithm to sort the blocks as in [51]. Finally, the client freshly encrypts the blocks and writes them back to the nodes in the path.

The stash is upper bounded by $O(\log N)$. The overall bandwidth may reach $\simeq 2.5 \log N$, for N blocks stored. In Ring ORAM, eviction happens periodically after a controllable parameter $A = 2z$ accesses where z is the number of blocks in each bucket [52].

3.2 Mapping an ORAM to a P2P setting

We start from a traditional ORAM in a client / server model where the client is trusted and the server is not, and simulate it on a tracker / peers setting. In particular, we consider that the server’s memory is organized in a tree structure, and we delegate every node in the tree to a peer. That is, a full binary tree of N leaves is now distributed among $N_p = 2N - 1$ peers (refer to Figure 1). In practice, many nodes can be delegated to many peers based on the storage capacity of each peer.

Contrary to the client / server setting where the client is the only one who can fetch, modify or add a block, in P2P, the peers can also request and add new blocks. In addition, the peers are volatile, i.e., many peers can join or leave the network. Moreover, from a security perspective, the network peers do not trust each other, and an adversarial peer can always be interested in finding out the block being retrieved by other peers. To avoid this, the tracker instructs the peers in a P2P system to save encrypted blocks in their local memory (different from the conventional BitTorrent model). Our construction ensures that the peer neither has the keys necessary to decrypt its storage nor can it collude with other adversarial peers to recover it. In this setting, we first present a strawman approach that guarantees our security goal but is restricted in terms of scalability.

3.3 OBLIVP2P-0 : Centralized Protocol

Almost all ORAM constructions are in a client / server setting and not designed for a P2P setting. A simple approach is to map the role of the trusted client in an

ORAM setting (refer to Figure 1) to the trusted tracker in a P2P system. The client in ORAM is simulated by the trusted tracker (storing the position map, private keys and the stash) and the server by the untrusted peers (storing the encrypted blocks). With such a mapping from an ORAM model to a P2P setting, a peer (initiator) can request for a resource to the tracker. To access a particular resource, the tracker fetches the blocks from a path in the tree and decrypts them to get the desired block. It then returns the requested resource to the initiator peer. This simple *plug-&-play* construction satisfies all our P2P security requirements.

In OBLIVP2P-0, the trusted tracker behaves as the client in traditional ORAM model. Whenever a peer requests a block, the tracker performs all the ORAM access work, and then sends the plaintext block to the initiator. The tracker downloads the path composed of a logarithmic number of nodes, writes back the path with a fresh re-encryption before routing the block to the initiator. As long as the tracker is trusted, this ensures the obliviousness property of peers’ accesses, as stated by definition 2.2.

Upload algorithm. To upload a file, the peer divides it into data blocks and sends the blocks to the tracker. The tracker appends the block to the stash stored locally while generating new random tags. The tracker updates accordingly TagMap, and FileMap (refer to Table 1).

Fetch algorithm. To fetch a file, the peer sends the file identifier, as an instance a filename, to the tracker. The tracker fetches from the FileMap and TagMap the corresponding blocks and sends requests to the corresponding peers to retrieve the blocks, following the Ring ORAM Access protocol. For every retrieved block, the tracker sends the plaintext block to the requesting peer.

Sync algorithm. The synchronization happens after every $A \simeq 2z$ accesses [52] (e.g., nearly 8 accesses) at which point the tracker evicts the stash.

Tracker as Bottleneck. In OBLIVP2P-0, the tracker has to transmit / encrypt a logarithmic number of blocks on every access. The tracker requires a bandwidth of $O(\log N \cdot B)$ where B is the block size and the computation cost of $O(\log N \cdot E)$ where E is time for encrypting / decrypting a block. Moreover, our evaluation in Section 5 shows that the eviction step is network-intensive. In a P2P setting with large number of accesses per second, the tracker creates a bottleneck in the network.

3.4 OBLIVP2P-0 Analysis

Our analysis follow from Ring ORAM construction. To access a block the tracker has to transmit $\sim 2.5 \log N \cdot B$ bits per access. During a block access or eviction, any peer at any time transmits $O(B)$ bits. The tracker’s main

computational time consists of decrypting and encrypting the stash. Since the stash has a size of $O(\log N)$ blocks, the tracker does $O(\log N)$ blocks encryption/decryption. In terms of storage, every peer has $(z + s)$ blocks to store, where z is number of real blocks and s is a parameter for dummy blocks. From a security perspective, it is clear that if there are two sequences verifying the constraints of Definition 2.2, a malicious peer monitoring their access pattern cannot infer the retrieved blocks, since after every access the block is assigned to a random path in the simulated ORAM.

4 OBLIVP2P-1: Distributed Protocol

In this section, we describe our main contribution, OBLIVP2P-1 protocol that provides both security and scalability properties. In designing such a protocol, our main goal is to avoid any bottleneck on the tracker i.e., none of the real blocks should route through the tracker for performing an access or evict operations of ORAM. We outline the challenges in achieving this property while still retaining the obliviousness in the network.

4.1 Challenges

First Attempt. A first attempt to reduce tracker’s overhead is to modify OBLIVP2P-0 such that the heavy computation of fetching the path of a tree and decrypting the correct block is offloaded to the initiator peer. On getting a resource request from a peer, the tracker simply sends information to the peer that includes the path of the tree to fetch, the exact position of the requested block and the key to decrypt it. However, unlike standard ORAM, the peer in our model is not trusted. Giving away the exact position of the block to the initiator peer leaks additional information about the requested resource in our model, as we explain next.

Recall that in a tree-based ORAM, blocks are distributed in the tree such that the recently accessed blocks remain in the top of the tree. In fact, after every eviction the blocks in the path are pushed down as far as possible from the root of the tree. As an instance, after N deterministic evictions, all blocks that were never accessed are (very likely) in the leaves. Conversely, consider that an adversarial peer makes two back-to-back accesses. In the first access, it retrieves a block from the top of the tree while in the second access it retrieves a block from a leaf. The adversarial peer (initiator) learns that the first block is a popular resource and is requested before by other peers while the second resource is a less frequently requested resource. This is a well known issue in tree-based ORAM, and is recently formulated as the *block history* problem [57]. Disclosing the block position, while hiding the scheme obliviousness requires to

address the block history challenge in ORAM. Unfortunately, an ORAM hides the block history only if the communication spent to access a block dominates the number of blocks stored in the entire ORAM. This would be asymptotically equivalent to downloading the entire ORAM tree from all the peers. We refer readers to [57] for more details.

Second Attempt. Our second attempt is a protocol that selects a block while *hiding* the block position from the adversary i.e., to hide which node on the path holds the requested block. Note that in a tree-based ORAM, disclosing the path does not break obliviousness, but leaking which node on the path holds the requested block is a source of leakage. One trick is to introduce a circuit, a set of peers from the P2P network, that will simulate the operations of a mixnet. That is, the peers holding the path of the tree send their content to the first peer in the circuit, who then applies a random permutation, adds a new encryption layer, and sends the permuted path to the second peer and so on. The tracker, who knows all the permutations, can send the final block position (unlinked from original position) to the initiator, along with the keys to decrypt the block. The mixing guarantees that the initiator does not learn the actual position of the block. We note that mixing used here is for only one accessed “path”, which is already randomized by ORAM. Hence, it is not susceptible to intersection attack discussed in Section 2.3. Finally, the initiator then peels off all layers of the desired block to output the plaintext block.

However, there is an important caveat remaining in using this method. Note that the initiator has the keys to peel off all the layers of encryption and hence it has access to the same encrypted block fetched from the path in the tree. Thus, it can determine which peer’s encrypted block was finally selected as the output of the mixnet. Hence, delegating the keys to the initiator boils down to giving her the block position. One might think of eliminating this issue by routing the block through the tracker to peel off all layers, but this will just make the tracker again a bottleneck.

So far, our attempts have shown limitations, but pointed out that there is a need to formally define the desired property. Considering a tracker, the initiator, and the peers holding the path, we seek a primitive that given a set of encrypted blocks, the initiator can get the desired plaintext block, while no entity can infer the block position but the tracker. We refer to this primitive as *Oblivious Selection* (*OblivSel*) and describe it next.

4.2 Oblivious Selection

4.2.1 Definitions

We define *OblivSel* and its properties as follows:

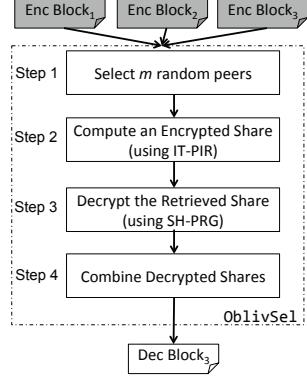


Figure 2: Oblivious Selection protocol using IT-PIR and Seed Homomorphic PRG as base primitives

Definition 4.1. (*Oblivious Selection*). *OblivSel* is a tuple of two probabilistic algorithms (*Gen*, *Select*) such that:

- $(\vec{\sigma}, \vec{r}) \leftarrow \text{Gen}(k, pos)$: a probabilistic algorithm run by the tracker, takes as input a key k and the block position pos , picks uniformly at random m peers (P_1, \dots, P_m) , and outputs $(\vec{\sigma}, \vec{r})$ where $\vec{\sigma} = \{\sigma_1, \dots, \sigma_m\}$ and $\vec{r} = \{r_1, \dots, r_m\}$ such that (σ_i, r_i) is given to the i th peer P_i .
- $\Delta \leftarrow \text{Select}(\vec{\sigma}, \vec{r}, \text{Enc}(k_1, \text{block}_1), \dots, \text{Enc}(k_L, \text{block}_L))$: a probabilistic algorithm run by m peers, takes as input $\vec{\sigma}$, \vec{r} , and a set of encrypted blocks $\text{Enc}(k_i, \text{block}_i)$, for $i \in [L]$, and outputs the value Δ .

Definition 4.2. *OblivSel*, is *correct*, if

$$\Pr[\forall pos \in [L], k \in \{0, 1\}^\lambda, (\vec{\sigma}, \vec{r}) \leftarrow \text{Gen}(k, pos); \Delta \leftarrow \text{Select}(\vec{\sigma}, \vec{r}, \text{Enc}(k_1, \text{block}_1), \dots, \text{Enc}(k_L, \text{block}_L)); \Delta = \text{Dec}(k, \text{Enc}(k_{pos}, \text{block}_{pos}))] = 1$$

For instance, if (Enc, Dec) is a private key encryption, *OblivSel* returns a decrypted block when the key given as input to the *Gen* function is the same as the private key of the block i.e., $\Delta = \text{block}_{pos}$ if $k = k_{pos}$.

Definition 4.3. (*Position Hiding*). We say that *OblivSel* is a position hiding protocol if for all probabilistic polynomial time global adversaries, including the initiator and the m peers, guess the position of the block pos with a negligible advantage in the implicit security parameter.

4.2.2 OblivSel Overview

The intuition for constructing *OblivSel* stems from the fact that the tracker cannot give the position or private key of the desired block to the peers in the network.

To privately select a block from the path without leaking its position, we propose to use an existing cryptographic primitive, called information-theoretical private

Structure	Mapping	Purpose
FileMap	file id fid <u>to</u> block addresses $\{\text{adr}_i\}_{i \in [\frac{f}{B}]}$	Blocks identification
TagMap	block address adr <u>to</u> tag $\overset{\$}{\leftarrow} [N_B]$	Path identification
NetMap	peer id pid <u>to</u> network info $(\text{IP}, \text{port}) \in \{0, 1\}^{128+16}$	Network representation
PosMap	block address adr <u>to</u> path and bucket position $\text{pos} \in [N_p] \times [L \cdot z + \text{stash}]$	Block exact localization
KeyMap	block address adr <u>to</u> key value $k \overset{\$}{\leftarrow} \mathbb{Z}_q$	Input of key block generation
StashList	peers' identifiers $\{\text{pid}_i\}_{i \in [\text{stash}]}$	Stash localization

Table 1: Various meta-information contained in the state s , for OBLIVP2P-0 and OBLIVP2P-1. B is the block size in bits, N_p the number of peers, N_B number of blocks, L the path length, and z the bucket size.

information retrieval (IT-PIR) [58]. IT-PIR requires a *linear* computation proportional to the data size that makes it expensive to use for real time settings. However, note that in our setting, we want to obliviously select a block from a logarithmic number of blocks (i.e., a path of the tree). Thus, applying IT-PIR over tree-based ORAM comes with significant computational improvement, hence making it practical to use in our protocol. The high level idea is to apply IT-PIR primitive only on one path since the obliviousness is already guaranteed by the underlined tree-based ORAM construction.

Figure 2 shows the steps involved in our OblivSel primitive. As a first step, the tracker randomly samples m peers from the network. For a bounded number of colluding adversarial peers in the system, this sample will contain at least one honest peer with high probability. The blocks of the path are fetched by all of the m peers. Each of the m peers then locally computes an encrypted share of the desired block using IT-PIR from the set of input blocks. Note that the tracker must not download the shares or it will violate our scalability requirement. On the other hand, we require to decrypt the block without giving away the private key to the network's peers. For this purpose, we make use of a second cryptographic primitive — a seed homomorphic pseudo-random generator (SH-PRG) [59]. The tracker generates a valid key share for each of the m peers to be used as seeds to the PRG function. Each peer decrypts (or unblinds) its encrypted share using its own key share such that the combination of decrypted shares results in a valid decryption of the original encrypted block in the tree. This property is ensured by SH-PRG and explained in detail in Section 4.2.3. Finally, each peer submits its decrypted share to the initiator peer who combines them to get the desired plaintext block. The colluding peers cannot recover the private key or the encrypted block since there is at least one honest peer who does not disclose its private information. This solves the issues raised in our second attempt.

Remark. OblivSel primitive can be used as a black box

Algorithm 1: IT-PIR protocol by Chor et al. [58]

```

1  $(r_1, \dots, r_m) \leftarrow \text{Query}(q, L, pos)$ 
    • randomly generate  $m - 1$  random vectors such that  $r_i \overset{\$}{\leftarrow} Z_q^L$ 
    • compute  $r_m$  such that for all  $j \in [L] \setminus \{pos\}$ , set
       $r_{m,j} = -\sum_{k=1}^{m-1} r_{k,j}$ , otherwise,  $r_{m,pos} = 1 - \sum_{k=1}^{m-1} r_{k,j}$ 
 $R_i \leftarrow \text{Compute}(r_i, DB)$ 
    • parse database such as  $DB = (\text{block}_1, \dots, \text{block}_L)$ 
    • compute  $R_i = \sum_{j=1}^L r_{i,j} \text{Block}_j$ 
 $\text{block}_{pos} \leftarrow \text{Recover}(R_1, \dots, R_m)$ : compute  $\text{block}_{pos} = \sum_{j=1}^m R_j$ 

```

in different settings such as distributed ORAMs to decrease the communication overhead. We further show in Section 4.2.4 that OblivSel is highly parallelizable and can leverage peers in the network such that the computation takes constant time.

4.2.3 Base Primitives

Information-theoretic PIR. Information-theoretic private information retrieval (IT-PIR) [58] is a cryptographic primitive that performs oblivious read operations while requiring multiple servers $m \geq 2$. In the following, we present the details of one of the first constructions of IT-PIR by Chor et al. [58] which is secure even when $m - 1$ among m servers collude passively, i.e., the servers collude in order to recover the retrieved block while not altering the protocol. An IT-PIR is a tuple of possibly randomized algorithms $\text{IT-PIR} = (\text{Query}, \text{Compute}, \text{Recover})$. Query takes as an input the block position pos to be retrieved, and outputs an IT-PIR query for m servers. Compute runs independently by every server, takes as input the corresponding IT-PIR query and outputs a share. Recover takes as inputs all shares output by all m servers, and outputs the plaintext block. We give the construction in Algorithm 1.

Seed homomorphic PRG (SH-PRG). A seed homomorphic PRG, G , is a pseudo-random generator over algebraic group with the additional property that if given

Algorithm 2: OblivSel with seed-homomorphic PRG

```

1  $(\vec{\sigma}, \vec{r}) \leftarrow \text{Gen}(k, pos)$ 
    • set  $\vec{r} = (r_1, \dots, r_m) \leftarrow \text{IT-PIR.Query}(q, L, pos)$ ;
    • set  $\vec{\sigma} = (\sigma_1, \dots, \sigma_m)$ , s.t.,  $(\sigma_1, \dots, \sigma_{m-1}) \xleftarrow{S} S^{m-1}$ , and
       $\sigma_m = k - \sum_{i=1}^{m-1} \sigma_i$ ;
    block  $\leftarrow \text{Select}(\vec{\sigma}, \vec{r}, DB)$  // Every peer  $P_i$ 
        • compute  $Eshare_i \leftarrow \text{IT-PIR.Compute}(r_i, DB)$ ;
        • set  $Dshare_i = Eshare_i - G(\sigma_i)$ ;
          // Initiator
        • compute  $\Delta = \sum_{i=1}^m Dshare_i$ ;

```

$G(s_1)$ and $G(s_2)$, then $G(s_1 \oplus s_2)$ can be computed efficiently. That is, if the seeds are in a group (S, \oplus) , and outputs in (G, \otimes) , then for any $s_1, s_2 \in S$, $G(s_1 \oplus s_2) = G(s_1) \otimes G(s_2)$. We refer to [60] for more details.

Decryption / Re-encryption using SH-PRG. Leveraging the property of SH-PRG, we explain the encryption, decryption and re-encryption of a block in our protocol. Every block in the tree is encrypted as $\text{Enc}(k_1, \text{block}) = \text{block} + G(k_1)$. The decryption of the block can be then represented as $\text{block} = \text{Dec}(k_1, \text{Enc}(k_1, \text{block})) = \text{block} + G(k_1) - G(k_1)$. For re-encrypting the encrypted block with a different key k_2 , the tracker decrypts the encrypted block with a new secret key of the form $k_1 - k_2$ such that, $\text{Dec}(k_1 - k_2, \text{Enc}(k_1, \text{block})) = \text{block} + G(k_1) - G(k_1 - k_2) = \text{block} + G(k_2) = \text{Enc}(k_2, \text{block})$.

4.2.4 OblivSel Instantiation

In the following, we present an instantiation of OblivSel. We consider a set of L encrypted blocks. Each block block_i is a vector of elements in a finite group G of order q . For every block, the key is generated at random from \mathbb{Z}_q . The tracker has to keep an association between the block key and its position. An algorithmic description is given in Algorithm 2.

The tracker runs the Gen algorithm, which takes as inputs the secret key k with which the block is encrypted and the block's position pos , and outputs a secret shared value of the key, $\vec{\sigma}$, as well as the IT-PIR queries, \vec{r} . Every peer P_i holds a copy of the L encrypted blocks and receives a share of the key, σ_i , as well as its corresponding query, r_i . Next, every peer runs locally an IT-PIR.Compute on the encrypted blocks and outputs a share, $Eshare_i$. After getting the encrypted share $Eshare_i$, each peer subtracts the evaluation of the SH-PRG G on σ_i from $Eshare_i$ ($Eshare_i - G(\sigma_i)$) to get the decrypted share $Dshare$. Finally, initiator outputs the sum of all the $Dshare_i$'s received from the m peers to

get the desired decrypted block. As long as there is one non-colluding peer among the m peers and G is a secure PRG, the scheme is *position hiding*.

Highly Parallelizable. Notice that, in Algorithm 2, each of the m peers performs scalar multiplications proportional to the number of encrypted input blocks. The encrypted blocks can be further distributed to different peers such that each peer performs constant number of scalar multiplications. Given the availability of enough peers in the network, OblivSel is extremely parallelizable and therefore provides a constant time computation.

OblivSel as a building block. OblivSel protocol can be used as a building block in our second and main OBLIVP2P-1. For fetching a block, an invocation of OblivSel is sufficient as it obviously selects the requested block and returns it in plaintext to the initiator. Additional steps such as re-encrypting the block and adding it to stash are required to complete the fetch operation. The details of these steps are in Section 4.3.

However, the eviction operation in ORAM poses an additional challenge. Conceptually, an eviction consists of block sorting, where the tracker re-orders the blocks in the path (and the stash). Fortunately, our protocol can perform eviction by several invocation of OblivSel primitive. Given the new position for each block, the P2P network can be instructed to invoke OblivSel recursively to output the new *sorted* path. The encryption of blocks has to be refreshed, but this is handled within OblivSel protocol itself when refreshing the key, using seed homomorphic PRG. We defer the concrete details of performing oblivious eviction to Section 4.3.

4.3 OBLIVP2P-1: Complete Design

In a P2P protocol for a content sharing system the tracker is responsible for managing the sharing of resources among the peers in the network. To keep a consistent global view on the network, the tracker keeps some *state* information that we formally define below:

Definition 4.4. *P2P network's state consists of: (1) number of possible network connections per peer, and (2) a lookup associating a resource to a (set of) peer identifier.*

The tracker can store more information in the state depending on the P2P protocol instantiating the network. We start first by formalizing a P2P protocol.

Definition 4.5. *A P2P protocol is a tuple of four (possibly interactive) algorithms $P2P = (\text{Setup}, \text{Upload}, \text{Fetch}, \text{Sync})$ involving a tracker, \mathcal{T} , and a set of peers, (P_1, \dots, P_n) , such that:*

- $s' \leftarrow \text{Setup}(s, \{\text{pid}\})$: run by the tracker \mathcal{T} , takes as inputs a state s and a (possibly empty) set of peers identifiers $\{\text{pid}\}$, and outputs an updated state s' .

Scheme	Tracker bandwidth (bits)	Network bandwidth (# blocks)	Tracker # encryption	Network computational overhead	Network Storage overhead	Tracker storage # blocks
OBLIVP2P-0	$O(\log N \cdot B)$	$O(1)$	$O(\log N \cdot E)$	—	$O(1)$	$O(\log N)$
OBLIVP2P-1	$O(\log^3 N)$	$O(\frac{\log N}{N})$	—	$O(\frac{\log^4 N}{N} \cdot \mathcal{E})$	$O(\text{burst})$	—

Table 2: Comparison of OBLIVP2P instantiation per access. B the block size, N the number of blocks in the network, E the overhead of a block encryption, \mathcal{E} a multiplication in elliptic curve group, burst the number of versions

- $(\text{out}, (A'_1, \dots, A'_m), s') \leftarrow \text{Upload}((\text{fid}, \text{file}), (A_1, \dots, A_m), s)$: is an interactive protocol between an initiator peer, a (possibly randomly selected) set of $m \geq 0$ peers, and a tracker \mathcal{T} . The initiator peer has as input a file identifier fid, and the file file, the peers' input is memory array A_i each, while for the tracker its state s. The initiator's output is $\text{out} \in \{\perp, \text{file}\}$, the peers output each a modified local memory A'_i , while the tracker outputs an updated state s' .
- $(\text{file}, \perp, s') \leftarrow \text{Fetch}(\text{fid}, (A_1, \dots, A_m), s)$: is an interactive protocol between an initiator peer, a (possibly randomly selected) set of $m \geq 0$ peers, and a tracker \mathcal{T} . The initiator peer has as input a file identifier fid, the peers' input is a memory array A_i each, while for the tracker its state s. The initiator outputs the retrieved file file, each peer gets \perp , while the tracker outputs an updated state s .
- $((A'_1, \dots, A'_m), s') \leftarrow \text{Sync}((A_1, \dots, A_m), s)$: is an interactive protocol between the tracker and a (possibly randomly selected) set of $m \geq 0$ peers. The peers' input is a memory array A_i each, while for the tracker its state s. The peers output each a (possibly) modified memory array A'_i , while the tracker outputs an updated state s' .

Note that a modification of a file already stored in the network is always considered as uploading a new file.

Setup Algorithm. In a P2P network, different peers have different storage capacities and hence we differentiate between the number of blocks, N_B , and the number of physical peers N_P . For this, we fragment the conceptual ORAM tree into smaller chunks where every peer physically handles a number of buckets depending on its local available storage. In addition, to keep a consistent global view on the network, the tracker keeps some *state* information. In OBLIVP2P-1, the state is composed of different meta-information that are independent of the block size: FileMap, PosMap, TagMap, NetMap, KeyMap, and StashMap. Table 1 gives more details about the metadata. The state also contains a counter recording the last eviction step, and $\sim \frac{B}{\log q}$ points sampled randomly from a q -order elliptic curve group \mathbb{G} to be used for DDH seed homomorphic PRG, where B is the block size. The number of points in the generator needs to be equal to those in the data block. These points are publicly known

Algorithm 3: Fetch(fid, s): OBLIVP2P-1 fetch operation

```

Input: file id fid, and state s
Output: file {block}, and updated state s
// Initiator requests tracker for a file
1 {adr} ← FileMap(fid);
2 for adr in {adr} do
3   (tag, pos) ← (TagMap(adr), PosMap(adr));
4   k ← KeyMap(adr);
5   compute  $(\vec{\sigma}, \vec{r}) := \text{OblivSel.Gen}(k, pos)$ ;
6   set  $A = (\text{stash}, \mathcal{P}(\text{tag}, 1), \dots, \mathcal{P}(\text{tag}, L))$ ;
// Initiator retrieves the block
7   compute block := OblivSel.Select( $\vec{\sigma}, \vec{r}, A$ );
// Re-encryption with a new secret
8   compute  $k \xleftarrow{\$} Z_q$ ;
9   compute  $(\vec{\sigma}, \vec{r}) := \text{OblivSel.Gen}(k, pos)$ ;
10  append  $\Delta := \text{OblivSel.Select}(\vec{\sigma}, \vec{r}, A)$  to the stash, and update state
    s;
11 end

```

to all peers in the network. The tracker randomly distributes the stash among the peers and records this information in the StashList.

Fetch Algorithm. The Fetch process is triggered when a peer requests a particular file. The tracker determines the block tag and position from its state for all the blocks composing the file. The m peers, the tracker, and the initiator runs OblivSel protocol such that the initiator retrieves the desired block. The OblivSel is invoked a second time to add a new layer to the retrieved block and send it to the peer who will hold the stash. The tracker updates its state, in particular, update KeyMap with the new key, update the PosMap with the exact position of the block in the network (in the stash), and TagMap with the new uniformly sampled tag. We provide an algorithmic description of the Fetch process in Algorithm 3.

Sync Algorithm. The Sync in OBLIVP2P-1 consists of: (1) updating the state of the network, but also, (2) evicting the stash. The tracker determines the path to be evicted, $\text{tag} = v \bmod 2^L$ and then fetches the position of all blocks in the stash and the path, $\mathcal{P}(\text{tag})$. The tracker then generates, based on the least common ancestor algorithm (LCA), a permutation π that maps every block in $A = (\text{stash}, \mathcal{P}(v \bmod 2^L, 1), \dots, \mathcal{P}(v \bmod 2^L, L))$ to its new position in A' , a new array that will replace the evicted path and the stash. The block $A[\pi(i)]$ will be mapped *obliviously* to $A'[i]$, for all $i \in [| \text{stash} | + z \cdot L]$. The oblivious mapping between A and A' is performed by invoking OblivSel between the tracker, the peers in

Algorithm 4: Sync(s): OBLIVP2P-1 sync operation

```
Input: tracker state  $s$ 
// Fetch necessary parameters
1  $v \leftarrow s$ ;
2  $\{adr\} \leftarrow PosMap^{-1}(v \bmod 2^L)$ ;
3 for  $adr$  in  $\{adr\}$  do
4   | set  $T = T \cup tag \leftarrow TagMap(adr)$ ;
5 end
6 set  $A = (\text{stash}, \mathcal{P}(v \bmod 2^L, 1), \dots, \mathcal{P}(v \bmod 2^L, L))$ ;
7 Initialize an array  $A'$ ,  $\pi \leftarrow LCA(T, v)$ ;
// tracker generates key shares
8 for  $l$  from 1 to  $L + |\text{stash}|$  do
9   | if  $\exists adr, l = PosMap(adr)$  then
10     |   | set  $k \leftarrow KeyMap(adr)$ ;
11     |   | set  $k'' = k' - k$ ,  $k' \xleftarrow{s} Z_q$ ;
12     |   | compute  $(\vec{\sigma}_l, \vec{r}_l) = \text{OblivSel.Gen}(k'', \pi(l))$ ;
13   | else
14     |   | set  $k'' \xleftarrow{s} Z_q$ , compute  $(\vec{\sigma}_l, \vec{r}_l) = \text{OblivSel.Gen}(k'', \pi(l))$ ;
15   | end
16 end
// Peers generate the new array  $A'$ 
17 for  $j$  from 1 to  $L + |\text{stash}|$  do
18   | set  $A'[j] = \text{OblivSel.Select}(\vec{\sigma}_j, \vec{r}_j, A)$ ;
19 end
20 for  $j \in [m]$ , send  $A'_j[1, \dots, |\text{stash}|]$  and  $A'_j[|\text{stash}| + 1, \dots, L]$  to peers in
 $\mathcal{P}(v)$  and the stash, and update state  $s$ ;
```

the path and m peers, $|\text{stash}| + z \cdot L$ times. Note that (1) the blocks in A' are encrypted with a freshly-generated key, and (2) the mapping is not disclosed to any peers in the path as long as there is one non-colluding peer. Refer to Algorithm 4 for more detail about the Sync algorithm.

Upload Algorithm. A peer can request the tracker to add a file. For this, the tracker selects uniformly at random a set of m peers. The peer sends the file in a form of blocks. Every block is secret shared such that every peer in the m peers receives a share. The tracker generates a secret unique to the block, k . The tracker secret shares k to the m peers. The peers evaluate a seed-homomorphic PRG on the received shares and add it to the block share. Finally, the block is appended to a randomly selected peer in the network to hold a part of the stash.

4.4 Optimization: Handling Bursts

OBLIVP2P-1 has a functional limitation inherited by ORAMs. Any access cannot be started unless the previous one has concluded². In our case, the tracker can handle fetching several blocks before starting the Sync operation. In our setting, we target increasing the P2P network throughput while leveraging the network storage and communication. In order to build a scalable system, we propose several optimizations.

O1: Replication. In Ring ORAM, $A = 3$ accesses can be performed before an eviction is required. To support A parallel accesses, we replicate every block A times in the tree. This absorbs the fetching access time and allows A

²We do not consider a multi-processor architectures as those considered in OPRAM literature [56].

simultaneous accesses, even for requests to the same resource. Additionally, we may replicate every block over A times on different peers, in case that the peer holding the block is offline due to churn, and cannot serve the block to the other peers. Lastly, the network operator can deploy multiple trackers to serve peers simultaneously, which leads to the throughput of OBLIVP2P-1 proportional to the number of trackers.

O2: Pipelining. While the eviction is highly parallelizable in OBLIVP2P-1, an eviction can take a considerable amount of time to terminate. If we denote by f the average number of fetch requests in the P2P network, and by t the time to perform an eviction, then the system can handle all the accesses if $t < \frac{1}{f}$. However, in practice $t > \frac{1}{f}$ and therefore the accesses will be queued and creates a bottleneck. To address this issue, we create multiple copies of the buckets that are run with different instances of OBLIVP2P-1 protocol which overlays on the same network. In the setup phase, every node creates l copies of its bucket space. Every bucket will be associated to different versions of OBLIVP2P-1 instantiations. For example, with replication we can handle A accesses in parallel on the (same) i th version of the buckets, but the upcoming accesses will be made on the $(i+1)$ th version. This will absorb the eviction time. To sum up, having different versions will increase the throughput of the system to $\frac{l}{f}$. In order to prevent pipeline stalls, we need to choose $l \geq t \cdot f$ in our implementation.

Another aspect (not considered for our implementation) for further optimizations in our versioning solution is to distribute the communication overhead of the peers in the network. In fact, the peers holding blocks at the higher level of the tree will be accessed more often compared to lower levels. In order to distribute the communication load on the network peers, peers' location can be changed for different versions such that: the peer at the i th level of the tree in the j th version will be placed at the $(L-i+1)$ th level of the tree in the $(j+1)$ th version.

O3: Parallelizing Computation across m Peers. The scalar multiplication in the elliptic curve is expensive and can easily delay the fetch and sync time. For this, we consider every peer in the OblivSel as a set of peers. Whenever there is a need to perform scalar multiplication over a path, several peers participate in the computation and only the representative of the set will perform the aggregation. This optimization speeds up the OblivSel to be proportional to the number of peers used to parallelize a single peer.

5 Implementation and Evaluation

Implementation. We implement a prototype of OBLIVP2P-0 and OBLIVP2P-1 in Python. The im-

plementation contains 1712 lines of code (LOC) for OBLIVP2P-0 and 3226 for OBLIVP2P-1 accounting to a total of 4938 lines measured using CLOC tool [61]. Our prototype implementation is open source and available online [34]. As our building block primitives, we implement the Ring ORAM algorithm, IT-PIR construction and seed-homomorphic PRG. For Ring ORAM, we have followed the parameters reported by authors [52]. Each bucket contains $z = 4$ blocks and $s = 5$ dummy blocks. The eviction occurs after every 3 accesses. The blocks in OBLIVP2P-0 are encrypted using AES-CBC with 256 bit key from the pycrypto library [62]. For implementing IT-PIR and seed homomorphic PRG in OBLIVP2P-1, we use the ECC library available in Python [63]. We use the NIST P-256 elliptic curve as the underlying group.

Experimental Setup. We use the DeterLab network testbed for our experiments [64]. It consists of 15 servers running Ubuntu 14.04 with dual Intel(R) Xeon(R) hexa-core processors running at 2.2 Ghz with 15 MB cache (24 cores each), Intel VT-x support and 24 GB of RAM. The tracker runs on a single server while each of the remaining servers runs approximately 2400 peers. Every peer process takes up to 4 – 60 MB memory which limits the maximum network size to 2^{14} peers in our experimental set up. The tracker is connected to a 128 MBps link and the peers in each server share a bandwidth link of 128 MBps as well. We simulate the bandwidth link following the observed BitTorrent traffic rate distribution reported in [65]. In our experimental setting, multiple peers are simulated on a single machine hence our reported results here are conservative. In the real BitTorrent setting, every peer has its own separate CPU.

Evaluation Methodology. To evaluate the scalability and efficiency of our system, we perform measurements for a) the overall throughput of the system b) the latency for Fetch and Sync operations and c) the data transferred through the tracker for both OBLIVP2P-0 and OBLIVP2P-1. All our results are the average of 50 runs with 95% confidence intervals for each of them. Along with the experimental results, we plot the theoretical bounds computed based on Table 2. This helps us to check if our experiments match our theoretical expectations. In addition, we perform separate experiments to demonstrate the effect of our optimizations on the throughput of our OBLIVP2P-1 protocol. For our experiments in this section, we leverage the technical optimization introduced in Section 4.4.

We vary the number of peers in the system from 2^4 to 2^{14} peers (capacity of our testbed) and extrapolate them to 2^{21} peers. Note that, when increasing the number of peers, we implicitly increase the total data size in the entire network which is computed as the number of peers \times the block size. That is, our P2P network handles a total

data size that spans from 16 KB to 32 GB. For our evaluation, we consider each peer holds one ORAM bucket because of the limited available memory. In reality, every peer can hold more buckets. Note that, we linearly extrapolate our curves to show the expected results for larger number of peers starting from $2^{15} - 2^{21}$ (shown dotted in the Figures), and therefore larger data size in the network. Aligned to the chunks in BitTorrent, we select our blocksize as 128 KB, 512 KB and 1 MB.

5.1 Linear Scalability with Peers

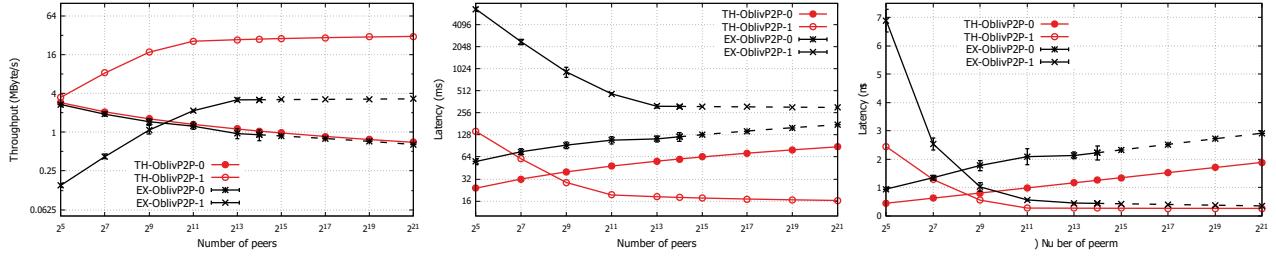
The throughput is an important parameter in designing a scalable P2P protocol. We define the throughput, as the number of bits that the system can serve per second.

From Figure 3a, we observe that the throughput of OBLIVP2P-0 decreases with the increase in the total number of peers in the network. For a network size of 2^{14} peers, the experimental maximum throughput is 0.91 MBps. As we extrapolate to larger network size, the maximum throughput decreases, e.g., for 2^{21} peers, the throughput is 0.64 MBps. This shows that as the network size increases, the tracker starts queuing the requests that will eventually lead to a saturation. However, for OBLIVP2P-1, the maximum throughput for network size of 2^{14} is 3.19 MBps and is 3.29 MBps when extrapolated to 2^{21} peers. The throughput increases as there are more peers available in the network to distribute the computation costs. The throughput shows a similar behaviour for blocksize of 128 KB and 1 MB (as shown in Figure 5). Hence, we expect OBLIVP2P-1 to provide better throughput in a real setting where more computational and communication capacity for each peer can be provisioned. The throughput values for OBLIVP2P-1 are calculated after applying all the 3 optimizations discussed in Section 4.4. The behaviour of the theoretical throughput matches our experimental results. The theoretical throughput has higher values as it does not capture the network latency in our test environment.

Result 1. Our results show that the centralized protocol is limited in scalability and cannot serve a large network. Whereas, the throughput for OBLIVP2P-1 linearly scales (0.15 – 3.39 MBps) with increasing number of peers ($2^5 - 2^{21}$) in the network.

Result 2. For a block of size 512 KB and 2^{14} peers, OBLIVP2P-1 serves around 7 requests / second which can be enhanced with multiple copies of ORAM trees in the network.

Remark. The throughput may be acceptable to privacy-conscious users (e.g., whistleblowers), where privacy concerns outweigh download / upload latencies. As long as the number of request initiators is small, the perceived throughput remains competitive with a non-



(a) The throughput for OBLIVP2P-1 linearly scales with the increase in network size.

(b) The latency for fetching a block for OBLIVP2P-1 reduces up to 2^{13} and then becomes constant.

(c) The latency for sync operation for OBLIVP2P-1 reduces up to 2^{13} and then becomes constant.

Figure 3: Theoretical (Th) and experimental (Ex) comparison of OBLIVP2P-0 and OBLIVP2P-1 parameters for block size of 512 KB

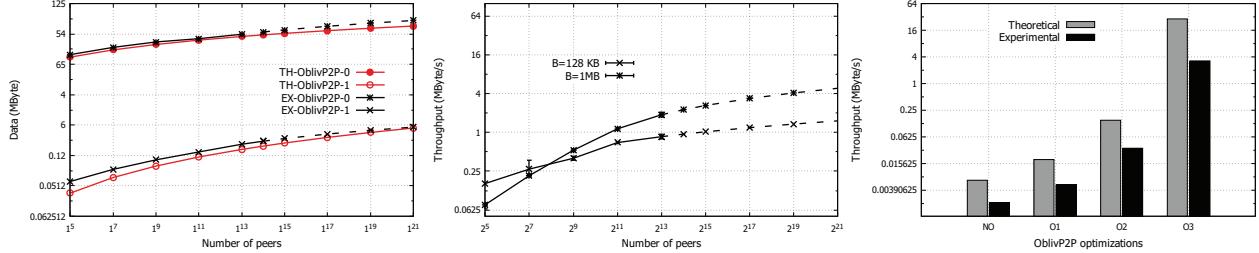


Figure 4: The data transferred through the tracker for OBLIVP2P-0 increases linearly with the number of peers

Figure 5: The throughput for blocksize 128 KB and 1 MB increases with increase in the network size.

Figure 6: Impact of optimizations (O1-O3) on the throughput of OBLIVP2P-1 for 2^{14} peers and blocksize of 512 KB.

oblivious P2P system. Further, the network operator can deploy multiple trackers to serve peers simultaneously, which leads to the throughput of OBLIVP2P-1 proportional to the number of trackers.

5.2 Latency Overhead and Breakdown

We define the latency as the time required to perform one ORAM operation in our P2P protocol. We measure the latency for the following operations:

Fetch. Figure 3b shows that the average time for fetching a block of 512 KB increases for OBLIVP2P-0 with increase in the size of the network. This is due to the increased computation and bandwidth overhead at the tracker. However, for OBLIVP2P-1, the latency initially reduces with the increasing number of peers (from 2^5 to 2^{11}) and then becomes constant after the network is large enough (around 2^{13}) to distribute the computation cost in the network³. OBLIVP2P-1 has a higher latency for fetch as compared to OBLIVP2P-0 due to the expensive computation required for performing scalar multiplication. The average time for fetching a block of size 512 KB is around 0.31 s for a network size of 2^{14} peers and

remains steady with increase in the number of peers.

Sync. We measure the time for performing a sync operation for different network sizes. Figure 3c shows that the time for performing a sync operation increases in OBLIVP2P-0 with increase in the number of peers. Whereas for OBLIVP2P-1, the sync time reduces gradually at first and then becomes steady after the network size reaches 2^{13} peers which is as expected through our theoretical calculation. OBLIVP2P-1 uses the peers in the network to distribute the computation load and hence the sync time tends to be steady for large network sizes.

Data transferred through tracker. Figure 4 shows the amount of data that is transferred through the tracker per request. We perform this measurement to show that the centralized tracker becomes a bottleneck in OBLIVP2P-0. The amount of data that the tracker has to process increases with increase in the number of peers. At 2^{21} peers, the amount of data is 118 MB (almost) reaching the bandwidth limit (128 MBps) of the tracker. Whereas, for OBLIVP2P-1 the amount of data transferred is around 1 MB for 2^{21} peers. This implies that the tracker could manage up to 128 copies of ORAM tree in parallel, which will increase the overall throughput by 128 times.

Result 3. OBLIVP2P has no centralized infrastructure as a bottleneck, ensuring that communication and computational overhead can be completely offloaded to the network.

³Since a large number of nodes (e.g., over 1000 nodes) share one physical machine, its limited computation power drastically affects our result. Therefore, to be more realistic, we use the ideal computing and decoding/encoding time for each node solely in one physical machine as the computing time per node, and simulate our experiments for OBLIVP2P-1.

5.3 Optimization Measurements

We perform incremental experiments to quantify the impact of each of the introduced optimizations on the overall throughput in Section 4.4, as shown in Figure 6. We fix the number of peers in the network to be equal to 2^{14} and the block size to 512 KB. We chose our optimization parameters based on our results in section 5.3. We fix the number of replicas to be equal to $A = 3$, i.e., the same data block is replicated three times. The burst parameter needs to be in $O(\frac{B}{\log q} \log N_p)$, where N_p is the number of peers, B the block size, and q the elliptic curve group order. Finally, we fix the number of parallel peers in the OblivSel.Select algorithm to be in $O(\frac{B}{\log q} \log N_p)$.

O1: Replication. Replication enables to perform $A = 3$ fetch operations in parallel. This implies that the throughput *theoretically* increases 3 times when compared to our baseline without any optimizations. Our experimental results show that we have 2.55 times improvement over the baseline, as expected theoretically.

O2: Pipelining. We evaluate the effect of our optimization (O2) that absorbs the eviction time by pipelining the fetch requests to different versions of the ORAM tree in the P2P network. We show that this optimization, when coupled to (O1), has theoretically increased the overall throughput by 23.05 times if compared to the baseline. Our experiments are aligned to our theoretical results and show 17.2 times improvement over the baseline with a burst parameter of 17. Clearly, if the number of versions increases beyond 17, then OBLIVP2P-1 can handle parallel accesses, hence increasing the system throughput.

O3: Parallelizing m peers. We measure the effect of parallelizing the computation load of m peers by leveraging more peers in the network on the overall throughput of the system. We increase the number of peers to 116 peers that are used to compute the fetch and sync operations. Our theoretical result shows an improvement of 4398 times over the baseline, when coupled with (O1) and (O2). Our experiments support this result and demonstrates 1589 times improvement, the difference is due to the real network latency are not considered in our theoretical calculation.

Result 4. OBLIVP2P-1 is subject to several optimizations due to its highly parallelizable design.

6 OBLIVP2P-1 Analysis

In this section, we present the theoretical analysis on computation / communication overhead of tracker / peers and security analysis for OBLIVP2P-1.

6.1 Performance

We report OBLIVP2P-1 computation and communication overhead for the tracker and the network in Table 2. In particular, OBLIVP2P-1’s tracker transmits a number of bits *independent* of the block size, the tracker does not perform any computation on the blocksize or store any block locally.

Tracker overhead. To fetch a block, the tracker invokes OblivSel twice. While for the eviction, the tracker performs OblivSel $(L \cdot z + |\text{stash}|)$ times. That is, it is sufficient to first analyze OblivSel overhead and than just conclude for the overall tracker overhead.

Within one instance of OblivSel, the tracker computes an IT – PIR.Query that outputs m vectors for m peers, each of size $L \cdot z + |\text{stash}|$. Each IT – PIR.Query vector costs $\log q(L \cdot z + |\text{stash}|)$ bits, where q is the group order. The tracker also needs to generate shares for the key, where the shares are in \mathbb{Z}_q . That is, one OblivSel costs the tracker $O(m \cdot \log q \cdot (L \cdot z + |\text{stash}|))$.

That is, the tracker has to transmit $O(m \cdot \log q \cdot (L \cdot z + |\text{stash}|)^2)$ bits. Considering $L, |\text{stash}| \in O(\log N)$, q the group order in $\text{poly}(N)$, m the number of peers and z the bucket size as constants, then the tracker needs to send $O(\log^3 N)$ bits independently of the block size. That is, if $\text{block} \in \Omega(\log^3 N)$, the tracker has a constant communication work per block. Moreover, the tracker is very lightweight as it does not perform any heavy computation such as encryption, decryption of blocks, which permits the tracker to handle frequent accesses.

Peers overhead. Considering the communication between the peers, the main communication overhead comes from block transfer from the peers holding the path to the selected m peers. The m peers are selected uniformly at random. Each peer receives $(z \cdot L + |\text{stash}|)$ blocks from the peers in the selected path and the stash. That is, in terms of communication overhead, the peers sends on average $\sum_{i=0}^L \frac{z}{2^i} + \frac{(z \cdot L + |\text{stash}|)}{N} + \frac{z}{N}$ blocks per peers in the network. Considering z constant and $L, |\text{stash}| \in O(\log N)$, implies that every peer is expected to transmit $O(\frac{\log N}{N})$ blocks per access.

In terms of computation, the main computational bottleneck consists of the scalar multiplication from the seed homomorphic PRG. For every OblivSel, every peer needs to perform $(z \cdot L + |\text{stash}|) \cdot \frac{B}{\log q}$ scalar multiplications per block. The second term, $\frac{B}{\log q}$, represents the number of points that a block contains. We also have $(z \cdot L + |\text{stash}|)$ instances of OblivSel during the eviction. That is, the total number of scalar multiplication equals $O((z \cdot L + |\text{stash}|)^2 \cdot \frac{B}{\log q})$. Finally, the amortized computation over the total number of peers in the network equals $O(\frac{\log^4 N}{N})$ multiplications per eviction, considering $B \in \Omega(\log^3 N)$ and $q \in \text{poly}(N)$.

6.2 Security Analysis

We show that OBLIVP2P-1 is an *oblivious* P2P as stated by Definition 2.2. For this, it is sufficient to show that an adversary cannot distinguish between a randomly generated string and the access pattern leaked by any peer’s real access. This underlines the fact that the access pattern is independent of the address of the requested block. In our threat model, the adversary can have access to the content of buckets, monitors the communication between the peers, and has a total view of the internal state of dishonest peers. Buckets’ content is assumed to be transmitted without any additional layer of encryption.

We present our *address-tag* experiment AT that captures our security definition. Let OBLIVP2P = (Setup, Upload, Fetch, Sync) represents an oblivious P2P protocol. Let $\mathcal{E} = (\text{Gen}, \text{Enc}, \text{Dec})$ be an IND\$ – CPA encryption scheme. Let \mathcal{G} be a secure pseudo-random generator. $\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}$ refers to the instantiation of the *address-tag* experiment by algorithms OBLIVP2P, \mathcal{E} , \mathcal{G} , and adversary \mathcal{A} . We denote by Col the event that m peers in the network collude and set $\Pr[\text{Col}] = \delta_m$, by \mathcal{B}_{δ_m} the Bernoulli distribution, and λ the security parameter.

In the following, we fix the number of colluding peers $c \in O(N^\varepsilon)$, for $0 < \varepsilon < 1$. We consider every peer in the network as a random variable distributed based on a Bernoulli distribution with probability equal to $\frac{c}{N} \in O(N^{\varepsilon-1})$. Let us denote by (X_1, \dots, X_m) the random variables of the selected peers for every instantiation of OblivSel. Note that $\Pr[\text{Col}] = \Pr[X_1 = 1 \text{ AND } \dots \text{ AND } X_m = 1]$. Since all X_i ’s are independent, then, $\Pr[\text{Col}] = \prod_{i=1}^m \Pr[X_i = 1] = (\frac{c}{N})^m$. That is, $\delta_m = (\frac{c}{N})^m$ which implies under our assumptions that $\delta_m \in O(2^{\log N \cdot m \cdot (\varepsilon-1)})$.

In the following experiment, we only consider the Fetch algorithm for obliviousness analysis. In our model, Upload sequences are indistinguishable by construction assuming that peers uploads blocks that are randomly distributed, and using random key for every block encryption. The experiment $\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, b)$ consists of:

- The adversary \mathcal{A} picks one access operation (Fetch, adr, \perp) and sends it to the challenger \mathcal{C}
- If $b = 1$, pick $X \xleftarrow{\mathcal{B}_{\delta_m}} \{0, 1\}$, if $X = 1$, then set var = adr, otherwise var = \perp and set

$$\begin{aligned} \pi_1 = & \{(\mathcal{P}(\text{tag}, 1), \dots, \mathcal{P}(\text{tag}, L)), \text{tag} \leftarrow \text{TagMap}[adr], \\ & (\text{Enc}(q_1), \dots, \text{Enc}(q_m)), \\ & (q_1, \dots, q_m) \leftarrow \text{IT-PIR.Query}(pos), \\ & pos \leftarrow \text{PosMap}[adr], \text{var}\} \end{aligned}$$

$$\begin{aligned} \text{If } b = 0, \text{ set } \pi_0 = & \{(P_1, \dots, P_L) \xleftarrow{\$} \mathbb{G}^{z \times L}, \\ & (q_1, \dots, q_m) \xleftarrow{\$} \{0, 1\}^{\lambda \times m}, \perp\} \end{aligned}$$

- Adversary \mathcal{A} has access to an oracle $\mathcal{O}^{\text{OblivP2P}}$ that issues the access patterns for polynomial number of accesses (while paths are re-encrypted for every request)
- \mathcal{A} outputs b'
- The output of the experiment is 1 if $b = b'$, otherwise 0. If $\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, b') = 1$, we say that \mathcal{A} won the experiment.

The experiment differentiates between a realistic setting where the adversary can see the access pattern, and in which a possible colluding setting can happen with a pre-fixed probability, δ_m , and an ideal setting where the adversary receives a random string. We slightly reformulate Definition 2.2 below.

Definition 6.1. We say that a P2P is oblivious iff for all PPT adversaries \mathcal{A} , there exists a negligible function negl such that:

$$\Pr[\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, 1) = 1] - \Pr[\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, 0) = 1] \leq \text{negl}(\lambda)$$

Theorem 6.1. If $\forall N > 1$, and $\forall \varepsilon < 1$, $\exists m > 1$ s.t. $2^{\log N \cdot m \cdot (1-\varepsilon)} \in \text{negl}(\lambda)$, \mathcal{G} is a secure pseudo-random generator, \mathcal{E} is IND\$ – CPA secure, then OBLIVP2P-1 is an oblivious P2P as in Definition 6.1.

Proof. To prove our theorem, we proceed with a succession of games as follows:

- Game₀ is exactly the same as $\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, 1)$
- Game₁ is the same as Game₀ except that the blocks in the buckets $\mathcal{P}(\text{tag}, i)$ are replaced with random points from \mathbb{G}
- Game₂ is the same as Game₁ except that the encrypted IT – PIR queries are replaced with random strings

From games’ description, we have

$$\Pr[\text{Game}_0 = 1] = \Pr[\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, 1) = 1], \quad (1)$$

For Game₁, we can build a distinguisher B_1 that reduces security of \mathcal{G} to PRG security such that:

$$\Pr[\text{Game}_0 = 1] - \Pr[\text{Game}_1 = 1] \leq \text{Adv}_{B_1, \mathcal{G}}^{\text{PRG}}(\lambda), \quad (2)$$

Similarly for Game₁, we can build a distinguisher B_2 that reduces \mathcal{E} to IND\$ – CPA security such that:

$$\Pr[\text{Game}_1 = 1] - \Pr[\text{Game}_2 = 1] \leq \text{Adv}_{B_2, \mathcal{E}}^{\text{IND\$-CPA}}(\lambda), \quad (3)$$

We need now to compute $\Pr[\text{Game}_2]$.

$$\begin{aligned} \Pr[\text{Game}_2] = & \Pr[\text{Col}] \cdot \Pr[\text{Game}_2 = 1 \mid \text{Col}] + \\ & \Pr[\overline{\text{Col}}] \cdot \Pr[\text{Game}_2 = 1 \mid \overline{\text{Col}}] \\ = & \delta_m + (1 - \delta_m) \frac{1}{N} \end{aligned}$$

On the other side $\Pr[\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, 0) = 1] = \frac{1}{N}$, since the tag is generated uniformly at random for every access.

$$\Pr[\text{Game}_2] - \Pr[\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, 0) = 1] = \delta_m(1 - \frac{1}{N}) \quad (4)$$

From equations 1, 2, 3, and 4 we obtain:

$$\begin{aligned} \Pr[\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, 1)] - \Pr[\text{AT}_{\mathcal{A}, \mathcal{E}, \mathcal{G}}^{\text{OblivP2P}}(\lambda, 0) = 1] &\leq \\ \delta_m(1 - \frac{1}{N}) + \text{Adv}_{B_2, \mathcal{E}}^{\text{INDS-CPA}} + \text{Adv}_{B_1, \mathcal{G}}^{\text{PRG}}. \end{aligned}$$

Since $\delta_m \in O(2^{\log N \cdot m \cdot (\varepsilon-1)})$, this ends our proof. \square

Quantitatively, if the number of peers in the network equals 2^{20} , number of colluding peers in the network is $c = N^{\frac{1}{2}}$ and $m = 12$, then $\delta_{12} = 2^{-120}$. Given the number of colluding peers and total number of peer, the value of m can always be adjusted to handle the desired colluding probability δ_m . In case of churn, the fraction c can vary and therefore the length of the circuit m has to be adapted to the new value. Furthermore, we implicitly assumed so far that no peer among the m selected leaves in the middle of the OblivSel process. If that occurs, the entire process has to abort, re-calculates the number of required peers m , and perform the OblivSel from scratch.

7 Discussion

Existing approaches. A valid question to investigate is whether existing solutions such as unlinkability or path non-correlation techniques can be extended to handle global adversaries and therefore prevent traffic analysis at the cost of providing more resources. It is easy to see that unlinkability techniques (e.g., mixnet) can provide better security in a P2P network under some assumptions. As an instance, assuming the case where a large number of peers behave as senders and issue requests that will be mixed by *sufficient* network peers before being answered by corresponding receivers' peers. Also, assuming that there is at least one honest peer in the mixing network, this solution would provide slightly the same level of security as OBLIVP2P where a global adversary cannot distinguish the senders' peers access pattern. However, this solution suffers from two downsides. First, there is a need to have *sufficient* number of senders' peers *on-line* in order to prevent intersection attacks. That is, in order to prevent traffic analysis, the number of senders represents a security parameter of the system that has to be maintained throughout the entire run of the system. Second, as the receivers' contents are theirs and are not encrypted, plus, all peers are considered honest-but-curious, a global adversary can easily find out what content is being accessed independently of the sender identity. This therefore does not achieve obliviousness as defined in our work but only a weaker version of it. On

the other hand, path non-correlation techniques conceptually cannot prevent against global adversary as we have detailed in Section 2. To sum up, it is not clear if existing techniques, even if given enough resources, can provide similar security insurances as those in OBLIVP2P.

Does better network & computation help? As empirically demonstrated in our evaluation section, the throughput of OBLIVP2P is around 3.19 MBps while considering only *one* tracker in the network. In a plain-text version of P2P system such as BitTorrent, the network leverages multiple trackers in order to handle more queries, and therefore increase the overall throughput. In OBLIVP2P, if we consider multiple copies of the entire network, we can also handle multiple trackers, and the throughput is expected to increase linearly with the number of trackers. However, as we delegate computation to the peers in OBLIVP2P, increasing the number of trackers beyond a particular threshold might turn out to be useless as the computation would represent a bottleneck of the system. As future work, we plan to investigate the asymptotic and empirical implications of including multiple trackers in the system. Moreover, it would be interesting to find out the relation between the number of trackers, number of peers for an ideal throughput of OBLIVP2P.

8 Related Work

Long-term traffic analysis. Anonymous systems like mix networks and onion routing are susceptible to long-term traffic analysis as shown in Section 2. Statistical disclosure attacks proposed by Danezis and enhanced by other researchers improve the likelihood of de-anonymizing users on these systems [66–73]. Moreover, existing traffic analysis attacks on onion routing based approaches [27–30, 45] can reveal users' identities with observing multiple communication rounds. Other P2P systems like Crowds [19], Tarzan [18], MorphMix [20], AP3 [21], Salsa [22], ShadowWalker [23], Freenet [3] offer anonymity for users. However, these systems show limits against global adversary with long-term traffic analysis capabilities.

Side-Channels. Previous work has shown possible attacks by leveraging side channels such as packet sizes, number of packets and timing. These side channels leak users' private information, e.g., illnesses/medications/surgeries, income and investment secrets [74]. An attacker can employ machine learning techniques (e.g., Support Vector Machines) on network traffic to identify the user's browsing websites [28–30, 45]. However, our focus in this paper is to only prevent long-term pattern traffic analysis. The aforementioned side-channels of traffic analysis are out of scope.

Multi-servers and parallel ORAM. There have been works on how to optimize ORAM constructions while leveraging multiple servers [75–77], multiple CPUs [56, 78], computational servers [53, 54], or distributed under a weaker threat model [55]. However, none of these recent constructions fit to a P2P setting as is. This is *mainly* due to the inherent client / server setting that results on a single entity bottleneck. The client has to either perform non-trivial computation or/and transmit several amount of bits. We briefly discuss these works below.

OblivStore [76], Lu and Ostrovsky [75], and Stefanov and Shi [77] demonstrate how to decrease the communication overhead while leveraging multiple ORAM nodes and servers. However, all these constructions are centralized and route the block through the tracker. This leads to a single entity bottleneck.

Recently, researchers have proposed oblivious parallel RAM (OPRAM) [56, 78]. This was motivated by current multi-cpu architectures that can access the same or multiple resources in parallel. However, OPRAM does not decrease the communication overhead making it as well a single-entity bottleneck. Dachman-Soled et al. introduced oblivious network RAM (ONRAM) [55]. ONRAM can reduce the communication overhead between the client and multiple banks of memory to be constant in the number of blocks. However, it assumes a weak threat model, and cannot achieve obliviousness in the case of a global adversary.

9 Conclusion

We advocate hiding data access patterns as a necessary step in defenses against long-term traffic pattern analysis in P2P content sharing systems. To this end, we propose OBLIVP2P— an oblivious peer-to-peer protocol. Our evaluation demonstrates that OBLIVP2P is parallelizable and linearly scalable with increase in number of peers, without bottleneck on a single entity.

Acknowledgement. We thank the anonymous reviewers of this paper for their helpful feedback. We also thank Erik-Oliver Blass, Travis Mayberry, Shweta Shinde and Hung Dang for useful discussions and feedback on an early version of the paper. This work is supported by the Ministry of Education, Singapore under Grant No. R-252-000-560-112. All opinions expressed in this work are solely those of the authors. A note of thanks to the DeterLab team [64] for enabling access to the infrastructure.

References

- [1] “Bittorrent,” <http://www.bittorrent.com/>.
- [2] “Storj.io,” <http://storj.io/>.
- [3] “Freenet: The free network,” <https://freenetproject.org>.
- [4] “Akamai,” <http://www.akamai.com/>.
- [5] S. Iyer, A. Rowstron, and P. Druschel, “Squirrel: A decentralized peer-to-peer web cache,” in *PODC*, 2002.
- [6] “Utorrent & bittorrent surge to 150 million monthly users,” <https://torrentfreak.com/bittorrent-surges-to-150-million-monthly-users-120109/>.
- [7] “Palo alto networks application usage & threat report,” <http://researchcenter.paloaltonetworks.com/app-usage-risk-report-visualization/>.
- [8] M. Piatek, T. Kohno, and A. Krishnamurthy, “Challenges and directions for monitoring p2p file sharing networks, or, why my printer received a dmca takedown notice,” in *HotSec*, 2008.
- [9] G. Siganos, J. M. Pujol, and P. Rodriguez, “Monitoring the bittorrent monitors: A bird’s eye view,” in *PAM*, 2009.
- [10] S. Le Blond, C. Zhang, A. Legout, K. Ross, and W. Dabbous, “I know where you are and what you are sharing: exploiting p2p communications to invade users’ privacy,” in *IMC*, 2011.
- [11] D. L. Chaum, “Untraceable electronic mail, return addresses, and digital pseudonyms,” *Communications of the ACM*, 1981.
- [12] G. Danezis, R. Dingledine, and N. Mathewson, “Mixminion: Design of a type iii anonymous remailer protocol,” in *IEEE S&P*, 2003.
- [13] U. Möller, L. Cottrell, P. Palfrader, and L. Sassaman, “Mixmaster protocol-version 2,” 2003.
- [14] M. G. Reed, P. F. Syverson, and D. M. Goldschlag, “Anonymous connections and onion routing,” *J-SAC*, 1998.
- [15] R. Dingledine, N. Mathewson, and P. F. Syverson, “Tor: The second-generation onion router,” in *USENIX Security*, 2004.
- [16] T. Wang, K. Bauer, C. Forero, and I. Goldberg, “Congestion-aware path selection for tor,” in *FC*, 2012.
- [17] M. Akhoondi, C. Yu, and H. V. Madhyastha, “Lastor: A low-latency as-aware tor client,” in *IEEE S&P*, 2012.
- [18] M. J. Freedman and R. Morris, “Tarzan: A peer-to-peer anonymizing network layer,” in *CCS*, 2002.
- [19] M. K. Reiter and A. D. Rubin, “Crowds: Anonymity for web transactions,” *TISSEC*, 1998.
- [20] M. Rennhard and B. Plattner, “Introducing morphmix: peer-to-peer based anonymous internet usage with collusion detection,” in *WPES*, 2002.
- [21] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach, “Ap3: Cooperative, decentralized anonymous communication,” in *SIGOPS European Workshop*, 2004.
- [22] A. Nambiar and M. Wright, “Salsa: a structured approach to large-scale anonymity,” in *CCS*, 2006.
- [23] P. Mittal and N. Borisov, “Shadowwalker: peer-to-peer anonymous communication using redundant structured topologies,” in *CCS*, 2009.
- [24] A. Pfitzmann and M. Hansen, “Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management—a consolidated proposal for terminology,” *Version v0*, 2008.
- [25] D. Agrawal and D. Kesdogan, “Measuring anonymity: The disclosure attack,” *IEEE S&P*, 2003.
- [26] D. Kesdogan and L. Pimenidis, “The hitting set attack on anonymity protocols,” in *IH*, 2004.
- [27] M. Edman and P. Syverson, “As-awareness in tor path selection,” in *CCS*, 2009.
- [28] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, “Peek-a-booo, i still see you: Why efficient traffic analysis countermeasures fail,” in *IEEE S&P*, 2012.

- [29] T. Wang and I. Goldberg, “Improved website fingerprinting on tor,” in *WPES*, 2013.
- [30] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, “Effective attacks and provable defenses for website fingerprinting,” in *USENIX Security*, 2014.
- [31] “Bittorrent over tor isn’t a good idea,” <https://blog.torproject.org/blog/bittorrent-over-tor-isnt-good-idea>.
- [32] S. L. Blond, P. Manils, C. Abdelberi, M. A. D. Kaafar, C. Castelluccia, A. Legout, and W. Dabbous, “One bad apple spoils the bunch: exploiting p2p applications to trace and profile tor users,” *arXiv*, 2011.
- [33] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious rams,” *J. ACM*, 1996.
- [34] “Oblivious peer-to-peer protocol,” <https://github.com/jiayaoqijia/OblivP2P-Code>.
- [35] “Gnutella,” <https://en.wikipedia.org/wiki/Gnutella>.
- [36] Y. Jia, G. Bai, P. Saxena, and Z. Liang, “Anonymity in peer-assisted cdns: Inference attacks and mitigation,” in *PETS*, 2016.
- [37] “Scaneye’s global bittorrent monitor,” <http://www.cogipas.com/anonymous-torrenting/torrent-monitoring/>.
- [38] K. Bauer, D. McCoy, D. Grunwald, and D. Sicker, “Bitstalker: Accurately and efficiently monitoring bittorrent traffic,” in *WIFS*, 2009.
- [39] T. Chothia, M. Cova, C. Novakovic, and C. G. Toro, “The unbearable lightness of monitoring: Direct monitoring in bittorrent,” in *SECURECOMM*, 2012.
- [40] G. Danezis and C. Diaz, “A survey of anonymous communication channels,” Tech. Rep., 2008.
- [41] H. Corrigan-Gibbs, D. Boneh, and D. Mazières, “Riposte: An anonymous messaging system handling millions of users,” in *IEEE S&P*, 2015.
- [42] M. Backes, A. Kate, S. Meiser, and E. Mohammadi, “(nothing else) mat or(s): Monitoring the anonymity of tor’s path selection,” in *CCS*, 2014.
- [43] “Tor suffers traffic confirmation attack,” <http://www.techtimes.com/articles/11711/20140802/tor-suffers-traffic-confirmation-attacks-say-goodbye-to-anonymity-on-the-web.htm>.
- [44] “Traffic confirmation attack,” <https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack>.
- [45] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, “Website fingerprinting in onion routing based anonymization networks,” in *WPES*, 2011.
- [46] J. Kong, W. Cai, and L. Wang, “The evaluation of index poisoning in bittorrent,” in *ICCSN*, 2010.
- [47] K. El Defrawy, M. Gjoka, and A. Markopoulou, “Bottorrent: Misusing bittorrent to launch ddos attacks.” *SRUTI*, 2007.
- [48] “Software Guard Extensions Programming Reference.” software.intel.com/sites/default/files/329298-001.pdf, 2013.
- [49] B. Parno, J. Howell, C. Gentry, and M. Raykova, “Pinocchio: Nearly practical verifiable computation,” in *IEEE S&P*, 2013.
- [50] E. Shi, T.-H. Chan, E. Stefanov, and M. Li, “Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost,” in *ASIACRYPT*, 2011.
- [51] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, “Path ORAM: an extremely simple oblivious RAM protocol,” in *CCS*, 2013.
- [52] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, “Constants Count: Practical Improvements to Oblivious RAM ,” in *USENIX Security*, 2014.
- [53] S. Devadas, M. van Dijk, C. Fletcher, L. Ren, E. Shi, and D. Wichs, “Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM,” *IACR*, 2015.
- [54] T. Moataz, T. Mayberry, and E.-O. Blass, “Constant Communication ORAM with Small Blocksize,” in *CCS*, 2015.
- [55] D. Dachman-Soled, C. Liu, C. Papamanthou, E. Shi, and U. Vishkin, “Oblivious network RAM and leveraging parallelism to achieve obliviousness,” in *ASIACRYPT*, 2015.
- [56] E. Boyle, K. Chung, and R. Pass, “Oblivious parallel RAM and applications,” in *TCC*, 2016.
- [57] D. S. Roche, A. J. Aviv, and S. G. Choi, “A practical oblivious map data structure with secure deletion and history independence,” *IACR*, 2015.
- [58] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, “Private information retrieval,” in *FOCS*, 1995.
- [59] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan, “Key homomorphic prfs and their applications,” in *CRYPTO*, 2013.
- [60] M. Naor and O. Reingold, “Number-theoretic constructions of efficient pseudo-random functions,” in *FOCS*, 1997.
- [61] “Cloc,” <http://cloc.sourceforge.net/>.
- [62] “Python cryptography toolkit,” <https://pypi.python.org/pypi/pycrypto>.
- [63] “Python ecc,” <https://github.com/john doe31415/joeecc>.
- [64] “Deterlab,” <https://www.isi.deterlab.net/index.php3>.
- [65] A. R. Bharambe, C. Herley, and V. N. Padmanabhan, “Analyzing and improving bittorrent performance,” *Microsoft Research*, 2005.
- [66] G. Danezis, “Statistical disclosure attacks,” in *Security and Privacy in the Age of Uncertainty*, 2003.
- [67] G. Danezis, C. Diaz, and C. Troncoso, “Two-sided statistical disclosure attack,” in *PETS*, 2007.
- [68] G. Danezis and A. Serjantov, “Statistical disclosure or intersection attacks on anonymity systems,” in *IH*, 2005.
- [69] N. Mathewson and R. Dingledine, “Practical traffic analysis: Extending and resisting statistical disclosure,” in *PETS*, 2005.
- [70] N. Mallesh and M. Wright, “The reverse statistical disclosure attack,” in *IH*, 2010.
- [71] C. Troncoso, B. Gierlichs, B. Preneel, and I. Verbauwhede, “Perfect matching disclosure attacks,” *LNCS*, 2008.
- [72] G. Danezis and C. Troncoso, “Vida: How to use bayesian inference to de-anonymize persistent communications,” in *PETS*, 2009.
- [73] F. Pérez-González and C. Troncoso, “Understanding statistical disclosure: A least squares approach,” in *PETS*, 2012.
- [74] S. Chen, R. Wang, X. Wang, and K. Zhang, “Side-channel leaks in web applications: A reality today, a challenge tomorrow,” in *IEEE S&P*, 2010.
- [75] S. Lu and R. Ostrovsky, “Distributed oblivious RAM for secure two-party computation,” in *TCC*, 2013.
- [76] E. Stefanov and E. Shi, “Oblivistore: High performance oblivious distributed cloud data store,” in *NDSS*, 2013.
- [77] ———, “Multi-cloud oblivious storage,” in *CCS*, 2013.
- [78] B. Chen, H. Lin, and S. Tessaro, “Oblivious parallel RAM: improved efficiency and generic constructions,” in *TCC*, 2016.

AuthLoop: Practical End-to-End Cryptographic Authentication for Telephony over Voice Channels

Bradley Reaves

University of Florida

reaves@ufl.edu

Logan Blue

University of Florida

bluel@ufl.edu

Patrick Traynor

University of Florida

traynor@cise.ufl.edu

Abstract

Telephones remain a trusted platform for conducting some of our most sensitive exchanges. From banking to taxes, wide swathes of industry and government rely on telephony as a secure fall-back when attempting to confirm the veracity of a transaction. In spite of this, authentication is poorly managed between these systems, and in the general case it is impossible to be certain of the identity (i.e., Caller ID) of the entity at the other end of a call. We address this problem with AuthLoop, the first system to provide cryptographic authentication solely within the voice channel. We design, implement and characterize the performance of an in-band modem for executing a TLS-inspired authentication protocol, and demonstrate its abilities to ensure that the explicit single-sided authentication procedures pervading the web are also possible on all phones. We show experimentally that this protocol can be executed with minimal computational overhead and only a few seconds of user time (≈ 9 instead of ≈ 97 seconds for a naïve implementation of TLS 1.2) over heterogeneous networks. In so doing, we demonstrate that strong end-to-end validation of Caller ID is indeed practical for all telephony networks.

1 Introduction

Modern telephony systems include a wide array of end-user devices. From traditional rotary PSTN phones to modern cellular and VoIP capable systems, these devices remain the de facto trusted platform for conducting many of our most sensitive operations. Even more critically, these systems offer the sole reliable connection for the majority of people in the world today.

Such trust is not necessarily well placed. Caller ID is known to be a poor authenticator [59, 18, 67], and yet is successfully exploited to enable over US\$2 Billion in fraud every year [28]. Many scammers simply block their phone number and exploit trusting users by

asserting an identity (e.g., a bank, law enforcement, etc.), taking advantage of a lack of reliable cues and mechanisms to dispute such claims. Addressing these problems will require the application of lessons from a related space. The Web experienced very similar problems in the 1990s, and developed and deployed the Transport Layer Security (TLS) protocol suite and necessary support infrastructure to assist with the integration of more verifiable identity in communications. While by no means perfect and still an area of active research, this infrastructure helps to make a huge range of attacks substantially more difficult. Unfortunately, the lack of similarly strong mechanisms in telephony means that *not even trained security experts can currently reason about the identity of other callers*.

In this paper, we address this problem with AuthLoop.¹ AuthLoop provides a strong cryptographic authentication protocol inspired by TLS 1.2. However, unlike other related solutions that assume Internet access (e.g., Silent Circle, RedPhone, etc [24, 73, 25, 5, 3, 6, 1, 74, 7]), accessibility to a secondary and concurrent data channel is not a guarantee in many locations (e.g., high density cities, rural areas) nor for all devices, mandating that a solution to this problem be network agnostic. Accordingly, AuthLoop is designed for and transmitted over the only channel certain to be available to all phone systems — audio. The advantage to this approach is that it requires no changes to any network core, which would likely see limited adoption at best. Through the use of AuthLoop, users can quickly and strongly identify callers who may fraudulently be claiming to be organizations including their financial institutions and their government [28].

We make the following contributions:

¹A name reminiscent of the “Local Loop” used to tie traditional phone systems into the larger network, we seek to tie modern telephony systems into the global authentication infrastructure that has dramatically improved transaction security over the web during the past two decades.

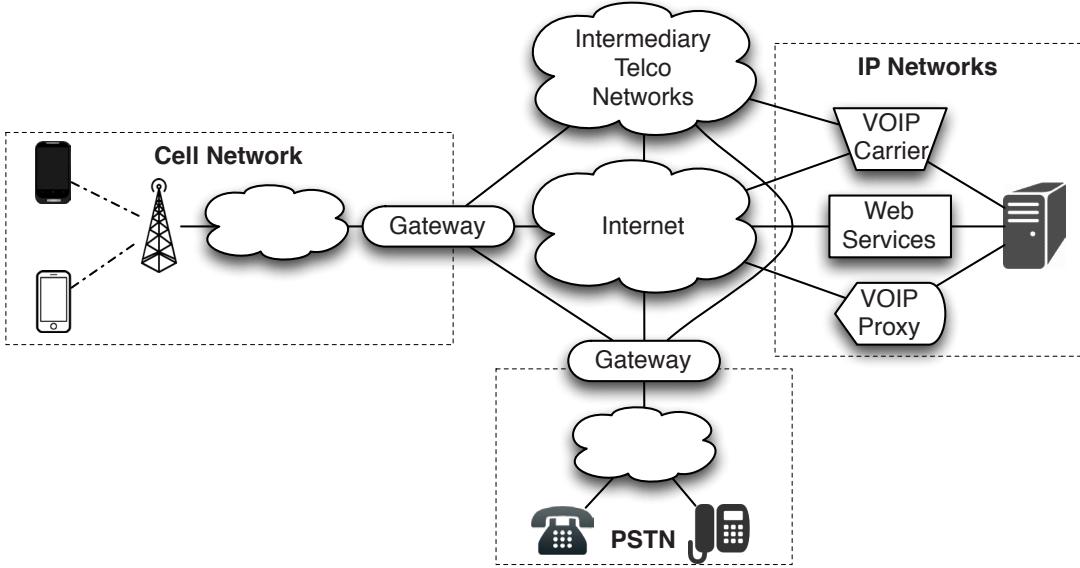


Figure 1: A high-level representation of modern telephony systems. In addition to voice being transcoded at each gateway, all identity mechanisms become asserted rather than attested as calls cross network borders. A strong end-to-end authentication must be designed aware of all such limitations.

- **Design a Complete Transmission Layer:** We design the first codec-agnostic modem that allows for the transmission of data across audio channels. We then create a supporting link layer protocol to enable the reliable delivery of data across the heterogeneous landscape of telephony networks.
- **Design AuthLoop Authentication Protocol:** After characterizing the bandwidth limitations of our data channel, we specify our security goals and design the AuthLoop protocol to provide explicit authentication of one party (i.e., the “Prover”) and optionally weak authentication of the second party (i.e., the “Verifier”).
- **Evaluate Performance of a Reference Implementation:** We implement AuthLoop and test it using three representative codecs — G.711 (for PSTN networks), AMR (for cellular networks) and Speex (for VoIP networks). We demonstrate the ability to create a data channel with a goodput of 500 bps and bit error rates averaging below 0.5%. We then demonstrate that AuthLoop can be run over this channel in an average of 9 seconds (which can be played below speaker audio), compared to running a direct port of TLS 1.2 in an average of 97 seconds (a 90% reduction in running time).

The remainder of this paper is organized as follows: Section 2 provides background information and related

work; Section 3 presents the details of our system including lower-layer considerations; Section 4 discusses our security model; Section 5 formally defines the AuthLoop protocol and parameterizes our system based on the modem; Section 6 discusses our prototype and experimental results; Section 7 provides additional discussion about our system; and Section 8 provides concluding remarks.

2 Background and Related Work

In this section, we provide an overview of modern telephony networks and review current and proposed practices of authentication in those networks.

2.1 Modern Telephony Networks

The landscape of modern telephony is complex and heterogeneous. Subscribers can receive service from mobile, PSTN and VoIP networks, and calls to those subscribers may similarly originate from networks implementing any of the above technologies. Figure 1 provides a high-level overview of this ecosystem.

While performing similar high-level functionality (i.e., enabling voice calls), each of these networks is built on a range of often incompatible technologies. From circuit-switched intelligent network cores to packet switching over the public Internet, very little information beyond the voice signal actually propagates across the borders of these systems. In fact, because many of these

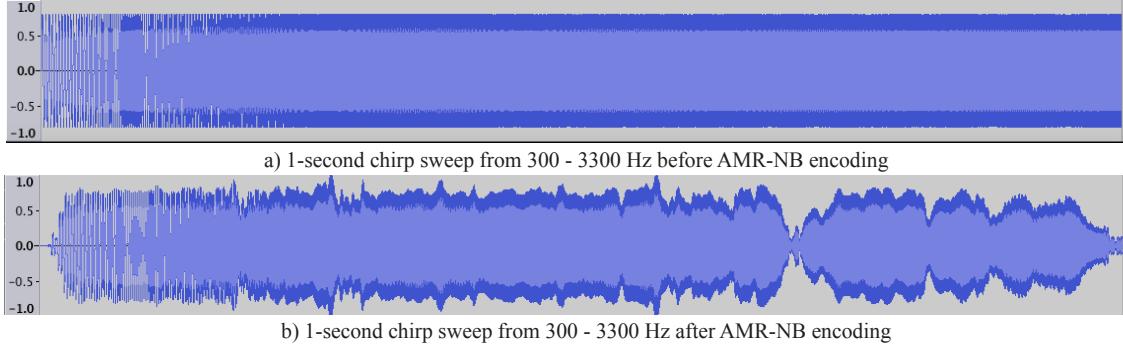


Figure 2: A comparison of a signal (a) before and (b) after being encoded with the AMR codec. Note that while the entirety of the signal is within the range of allowable frequencies for call audio, the received signal differs significantly from its original form. It is therefore critical that a high-fidelity mechanism for delivering data over a mobile audio channel be designed.

networks rely on different codecs for encoding voice, one of the major duties of gateways between these systems is the transcoding of audio. Accordingly, voice encoded at one end of a phone call is unlikely to have the same (or even similar) bitwise representation when it arrives at the client side of the call. As evidence, the top plot of Figure 2 shows a sweep of an audio signal from 300 to 3300 Hz (all within the acceptable band) across 1 second. The bottom plot shows the same signal after it has been encoded using the Adaptive Multi-Rate (AMR) audio codec used in cellular networks, resulting in a dramatically different message. This massive difference is a result of the voice-optimized audio codecs used in different telephony networks. Accordingly, successfully performing end-to-end authentication will require careful design for this non-traditional data channel.

One of the few pieces of digital information that can be optionally passed between networks is the Caller ID. Unfortunately, the security value of this metadata is minimal — such information is asserted by the source device or network, but never validated by the terminating or intermediary networks. As such, an adversary is able to claim any phone number (and therefore identity) as its own with ease. This process requires little technical sophistication, can be achieved with the assistance of a wide range of software and services, and is the enabler of greater than US\$2 Billion in fraud annually [28].

2.2 Authentication in Telephony Networks

Authentication has been the chief security concern of phone networks since their inception because of its strong ties to billing [69]. Little effort was taken for authentication in traditional landline networks as detecting billable activity on a physical link limited the scalability of attacks. First generation (1G) cellular systems were the first to consider such mechanisms given the multi-

user nature of wireless spectrum. Unfortunately, 1G authentication relied solely on the plaintext assertion of each user’s identity and was therefore subject to significant fraud [53]. Second generation (2G) networks (e.g., GSM) designed cryptographic mechanisms for authenticating users to the network. These protocols failed to authenticate the network to the user and lead to a range of attacks against subscribers [44, 26, 19, 68]. Third and fourth generation (3G and 4G) systems correctly implement mutual authentication between the users and providers [11, 12, 13]. Unfortunately, all such mechanisms are designed to allow accurate billing, and do little to help users identify other callers.

While a number of seemingly-cellular mechanisms have emerged to provide authentication between end users (e.g., Zphone, RedPhone) [24, 73, 25, 5, 3, 6, 1, 74, 7, 31, 30], these systems ultimately rely on a data/Internet connection to work, and are themselves vulnerable to a number of attacks [63, 52]. Accordingly, there remains no end-to-end solution for authentication *across voice* networks (i.e., authentication with any non-VoIP phone is not possible).

Mechanisms to deal with such attacks have had limited success. Websites have emerged with reputation data for unknown callers [2]; however, these sites offer no protection against Caller-ID spoofing, and users generally access such information after such a call has occurred. Others have designed heuristic approaches around black lists [4], speaker recognition [71, 16, 72, 17, 66, 41], channel characterization [18, 54], post hoc call data records [58, 47, 23, 40] and timing [61]. Unfortunately, the fuzzy nature of these mechanisms may cause them to fail under a range of common conditions including congestion and evasion.

Authentication between entities on the Internet generally relies on the use of strong cryptographic mecha-

nisms. The SSL/TLS suite of protocols are by far the most widely used, and help provide attestable identity for applications as diverse as web browsing, email, instant messaging and more. SSL/TLS are not without their own issues, including a range of vulnerabilities across different versions and implementations of the protocols [48, 27, 75, 34], weaknesses in the model and deployment of Certificate Authorities [57, 36, 37, 38, 29, 39, 20], and usability [55, 32, 60, 35, 65, 14, 15]. Regardless of these challenges, these mechanisms provide more robust means to reason about identity than the approaches used in telephony.

Telephony can build on the success of SSL/TLS. However, these mechanisms can not simply be built on top of current telephony systems. Instead, and as we will demonstrate, codec-aware protocols that are optimized for the limited bitrate and higher loss of telephony systems must be designed.

3 Voice Channel Data Transmission

To provide end-to-end authentication across any telephone networks, we need a way to transfer data over the voice channel. The following sections detail the challenges that must be addressed, how we implemented a modem that provides a base data rate of 500bps, and how we developed a link layer to address channel errors. We conclude with a discussion of what these technical limitations imply for using standard authentication technologies over voice networks.

3.1 Challenges to Data Transmission

Many readers may fondly remember dial-up Internet access and a time when data transmission over voice channels was a common occurrence. In the heyday of telephone modems, though, most voice channels were connected over high-fidelity analog twisted pair. Although the voice channel was band limited and digital trunks used a low sample rate of 8kHz, the channel was quite “well behaved” from a digital communications and signal processing perspective.

In the last two decades, telephony has been transformed. Cellular voice and Internet telephony now comprise a majority of all voice communications; they are not just ubiquitous, they are unavoidable. While beneficial from a number of perspectives, one of the drawbacks is that both of these modalities rely on heavily compressed audio transmission to save bandwidth. These compression algorithms – audio codecs – are technological feats, as they have permitted cheap, acceptable quality phone calls, especially given that they were developed during eras when computation was expensive. To do this, codec

designers employed a number of technical and psychoacoustic tricks to produce acceptable audio to a human ear, and these tricks resulted in a channel poorly suited for (if not hostile to) the transmission of digital data. As a result, existing voice modems are completely unsuited for data transmission in cellular or VoIP networks.

Voice codecs present several challenges to a general-purpose modem. First, amplitudes are not well preserved by voice codecs. This discounts many common modulation schemes, including ASK, QAM, TCM, and PCM. Second, phase discontinuities are rare in speech, and are not effective to transmit data through popular voice codecs. This discounts PSK, QPSK, and other modulation schemes that rely on correct phase information. Furthermore, many codecs lose phase information on encoding/decoding audio, preventing the use of efficient demodulators that require correct phase (i.e., coherent demodulators). Because of the problems with amplitude and phase modulation, frequency-shift modulation is the most effective technique for transmitting data through voice codecs. Even so, many codecs fail to accurately reproduce input frequencies — even those well within telephone voicebands (300–3400 Hz). Our physical layer protocol addresses these challenges.

3.2 Modem design

The AuthLoop modem has three goals:

1. Support highest bitrate possible
2. At the lowest error rate possible
3. In the presence of deforming codecs

We are not the first to address transmission of data over lossy compressed voice channels. Most prior efforts [70, 51, 42] have focused on transmission over a single codec, though one project, Hermes [33] was designed to support multiple cellular codecs. Unfortunately, that project only dealt with the modulation scheme, and did not address system-level issues like receiver synchronization. Furthermore, the published code did not have a complete demodulator, and our own implementation failed to replicate their results. Thus, we took Hermes as a starting point to produce our modem.

Most modems are designed around the concept of modulating one or more parameters — amplitude, frequency, and/or phase — of one or more sine waves. Our modem modulates a single sine wave using one of three discrete frequencies (i.e. it is a frequency shift key, or FSK, modem). The selection of these frequencies is a key design consideration, and our design was affected by three design criteria.

First, our modem is designed for phone systems, so our choice of frequencies are limited to the 300–3400Hz

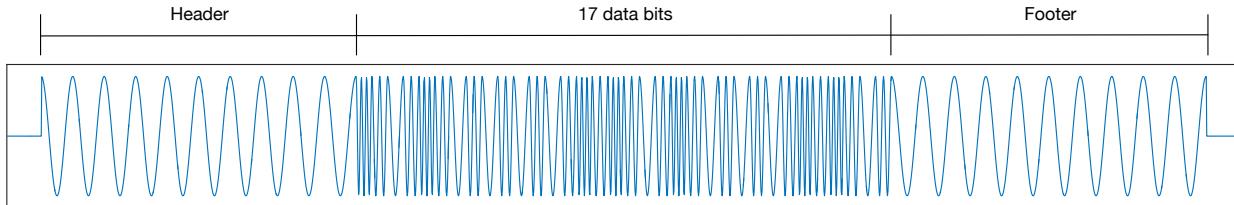


Figure 3: This 74ms modem transmission of a single frame demonstrates how data is modulated and wrapped in headers and footers for synchronization.

range because most landline and cellular phones are limited to those frequencies. Second, because we cannot accurately recover phase information for demodulation, our demodulation must be decoherent; the consequence is that our chosen frequencies must be separated by at least the symbol transmission rate [64]. Third, each frequency must be an integer multiple of the symbol frequency. This ensures that each symbol completes a full cycle, and it also ensures that each cycle begins and ends on a symbol boundary. This produces a continuous phase modulation, and it is critical because some voice codecs will produce artifacts or aliased frequencies in the presence of phase discontinuities. These constraints led to the selection of a 3-FSK system transmitting symbols at 1000 Hz using frequencies 1000, 2000, and 3000 Hz.

Unfortunately, 3-FSK will still fail to perform in many compressed channels simply because those channels distort frequencies, especially frequencies that change rapidly. To mitigate issues with FSK, we use a differential modulation: bits are encoded not as individual symbols, but by the relative difference between two consecutive symbols. For example, a “1” is represented by an increase in two consecutive frequencies, while a “0” is represented by a frequency decrease. Because we only have 3 frequencies available, we have to limit the number of possible consecutive increases or decreases to 2. Manchester encoding, where each bit is expanded into two “half-bits” (e.g. a “1” is represented by “10”, and “0” represented by “01”) limits the consecutive increases or decreases within the limit.

While these details cover the transmission of data, there are a few practical concerns that must be dealt with. Many audio codecs truncate the first few milliseconds of audio. In speech this is unnoticeable, and simplifies the encoding. However, if the truncated audio carries data, several bits will be lost every transmission. This effect is compounded if voice activity detection (VAD) is used (as is typical in VoIP and cellular networks). VAD distinguishes between audio and silence, and when no audio is recorded in a call VAD indicates that no data should be sent, saving bandwidth. However, VAD adds an additional delay before voice is transmitted again.

To deal with early voice clipping by codecs and VAD, we add a 20 ms header and footer at the end of each packet. This header is a 500 Hz sine wave; this frequency is orthogonal to the other 3 transmission frequencies, and is half the symbol rate, meaning it can be used to synchronize the receiver before data arrives. A full modem transmission containing 17 bits of random data can be seen in Figure 3.

To demodulate data, we must first detect that data is being transmitted. We distinguish silence and a transmission by computing the energy of the incoming signal using a short sliding window (i.e., the short-time energy). Then we locate the header and footer of a message to locate the beginning and end of a data transmission. Finally, we compute the average instantaneous frequency for each half-bit and compute differences between each bit. An increase in frequency indicates 1, a decrease indicates 0.

3.3 Link Layer

Despite a carefully designed modem, reception errors will still occur. These are artifacts created by line noise, the channel codec, or an underlying channel loss (e.g., a lost IP packet). To address these issues, we developed a link layer to ensure reliable transmission of handshake messages. This link layer manages error detection, error correction, frame acknowledgment, retransmission, and reassembly of fragmented messages.

Because error rates can sometimes be as high as several percent, a robust retransmission scheme is needed. However, because our available modem data rate is so low, overhead must be kept to a minimum. This rules out most standard transmission schemes that rely on explicit sequence numbers. Instead, our data link layer chunks transmitted frames into small individual blocks that may be checked and retransmitted if lost. We are unaware of other link layers that use this approach. The remainder of this subsection motivates and describes this scheme.

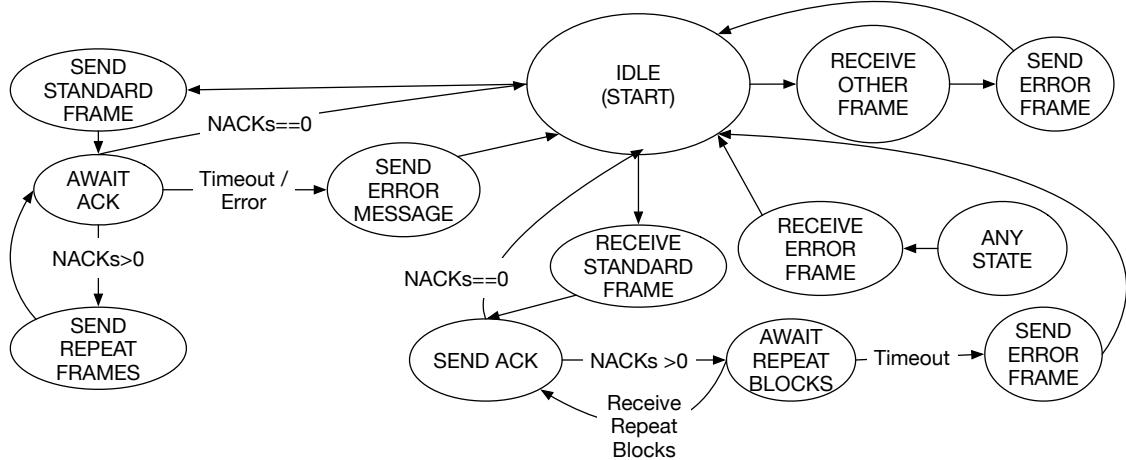


Figure 4: Link Layer State Machine

3.4 Framing and error detection

Most link layers are designed to transmit large (up to 12,144 bits for Ethernet) frames, and these channels either use large (e.g., 32-bit) CRCs² for error detection to retransmit the entire frame, or use expensive but necessary error correcting schemes in lossy media like radio. Error correcting codes recover damaged data by transmitting highly redundant data, often inflating the data transmitted by 100% or more. The alternative, sending large frames with a single CRC, was unlikely to succeed. To see why, note that:

$$P(\text{CorrectCRC}) = (1 - P(\text{biterror}))^{\text{CRClength}} \quad (1)$$

For a 3% bit error rate, the probability of just the CRC being undamaged is less than 38% — meaning two thirds of packets will be dropped for having a bad CRC independent of other errors. Even at lower loss rates, retransmitting whole frames for a single error would cause a massive overhead.

Instead, we divide each frame into 32-bit “blocks”. Each block carries 29 bits of data and a 3-bit CRC. This allows short sections of data to be checked for errors individually and retransmitted, which is closer to optimal transmission. Block and CRC selection was not arbitrary, but rather the result of careful modeling and analysis. In particular, we aimed to find an optimal tradeoff between overhead (i.e., CRC length) and error detection. Intuitively, longer CRCs provide better error detection and reduce the probability of an undetected error. More formally, a CRC of length l can guarantee detection of up to

²A Cyclic Redundancy Check (CRC) is a common checksum that is formed by representing the data as a polynomial and computing the remainder of polynomial division. The polynomial divisor is a design parameter that must be chosen carefully.

HD bit errors³ in a B -length block of data, and can detect more than HD errors probabilistically [43].

The tradeoff is maximizing the block size and minimizing the CRC length while minimizing the probability of a loss in the frame or the probability of an undetected error, represented by the following equations:

$$Pr(\text{lost frame}) = 1 - Pr(\text{successful frame}) \quad (2)$$

$$= 1 - (1 - p)^B \quad (3)$$

$$Pr\left(\frac{\text{undetected}}{\text{error}}\right) = 1 - \sum_{i=0}^{HD} \binom{B}{i} p^i (1-p)^{B-i} \quad (4)$$

where p represents the probability of a single bit error. The probability of undetected error is derived from the cumulative binomial distribution. Using these equations and the common bit error rate of 0.3% (measured in Section 6), we selected 32-bit blocks with a 3-bit CRC. We chose the optimal 3-bit CRC polynomial according to Koopman and Chakravarty [43]. These parameters give a likelihood of undetected error of roughly 0.013%, which will rarely affect a regular user. Even a call center user would see a protocol failure due to bit error once every two weeks, assuming 100 calls per day.

3.5 Acknowledgment and Retransmission

Error detection is only the first step of the error recovery process, which is reflected as a state machine in Figure 4.

When a message frame is received, the receiver computes which blocks have an error and sends an acknowledgment frame (“ACK”) to the transmitter. The ACK frame contains a single bit for each block transmitted to indicate if the block was received successfully or not.

³The Hamming distance of the transmitted and received data

Blocks that were negatively acknowledged are retransmitted; the retransmission will also be acknowledged by the receiver. This process will continue until all original blocks are received successfully.

By using a single bit of acknowledgment for each block we save the overhead of using sequence numbers. However, even a single bit error in an ACK will completely desynchronize the reassembly of correctly received data. Having meta-ACK and ACK retransmission frames would be unwieldy and inelegant. Instead, we transmit redundant ACK data as a form of error correction; we send ACK data 3 times in a single frame and take the majority of any bits that conflict. The likelihood of a damaged ACK is then:

$$\text{Block Count} \times 3 \times \Pr(\text{biterr})^2 \quad (5)$$

instead of

$$1 - (1 - \Pr(\text{biterr}))^{\text{Block Count}} \quad (6)$$

Note that there are effectively distinct types of frames – original data, ACK data, retransmission data, and error frames. We use a four-bit header to distinguish these frames; like ACK data, we send three copies of the header to ensure accurate recovery. We will explore more robust error correcting codes in future work.

3.6 Naïve TLS over Voice Channels

With a modem and link layer design established, we can now examine how a standard authentication scheme — TLS 1.2 — would fare over a voice channel.

Table 1 shows the amount of data in the TLS handshakes of four popular Internet services: Facebook, Google, Bank of America, and Yahoo. These handshakes require from 41,000 to almost 58,000 bits to transmit, and this excludes application data and overhead from the TCP/IP and link layers. At 500 bits per second (the nominal speed of our modem), these transfers would require 83–116 seconds *as a lower bound*. From a usability standpoint, standard TLS handshakes are simply not practical for voice channels. Accordingly, a more efficient authentication protocol is necessary.

4 Security Model

Having demonstrated that data communication is possible but extremely limited via voice channels, we now turn our attention to defining a security model. The combination of our modem and this model can then be used to carefully design the AuthLoop protocol.

The goal of AuthLoop is to mitigate the most common enabler of phone fraud: claiming a false identity via Caller ID spoofing. This attack generally takes the

Table 1: TLS Handshake Sizes

Site Name	Total Bits	Transmission Time (seconds at 500bps)
Facebook	41 544	83.088
Google	42 856	85.712
Bank of America	53 144	106.288
Yahoo	57 920	115.840
Average	48 688	97.732

form of the adversary calling the victim user and extracting sensitive information via social engineering. The attack could also be conducted by sending the victim a malicious phone number to call (e.g., via a spam text or email). An adversary may also attempt to perform a man in the middle attack, calling both the victim user and a legitimate institution and then hanging up the call on either when they wish to impersonate that participant. Finally, an adversary may attempt to perform a call forwarding attack, ensuring that correctly dialed numbers are redirected (undetected to the caller) to a malicious endpoint.

We base our design on the following assumptions. An adversary is able to originate phone calls from any telephony device (i.e., cellular, PSTN, or VoIP) and spoof their Caller ID information to mimic any phone number of their choosing. Targeted devices will either display this spoofed number or, if they contain a directory (e.g., contact database on a mobile phone), a name associated or registered with that number (e.g., “Bank of America”). The adversary can play arbitrary sounds over the audio channel, and may deliver either an automated message or interact directly with the targeted user. Lastly, the adversary may use advanced telephony features such as three-way calling to connect and disconnect parties arbitrarily. This model describes the majority of adversaries committing Caller ID fraud at the time of this work.

Our scenario contains two classes of participants, a Verifier (i.e., the user) and Prover (i.e., either the attacker of the legitimate identity owner). The adversary is active and will attempt to assert an arbitrary identity. As is common on the Web, we assume that Provers have certificates issued by their service provider⁴ containing their public key and that Verifiers may have weak credentials (e.g., account numbers, PINs, etc) but do not have certificates. We seek to achieve the following security goals in the presence of this adversary:

1. **(G1) Authentication of Prover:** The Verifier should be able to explicitly determine the validity of an asserted Caller ID and the identity of the Prover without access to a secondary data channel.

⁴See Section 7 for details.

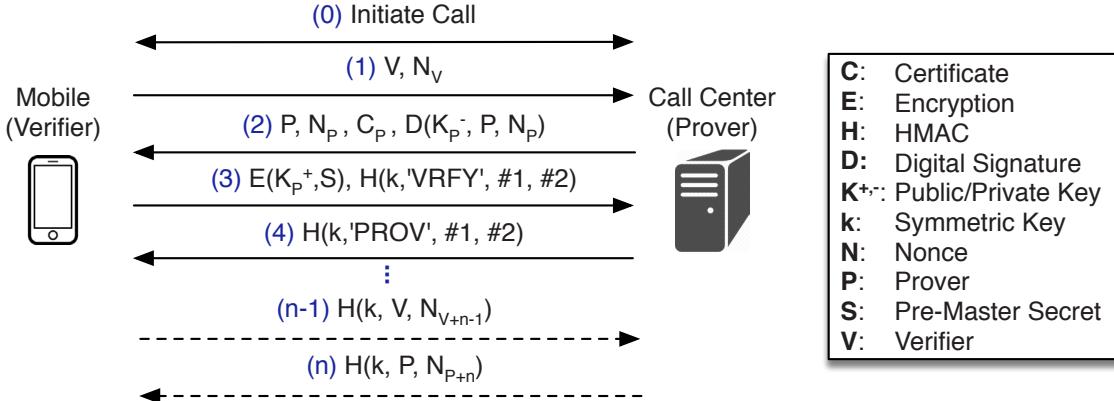


Figure 5: The AuthLoop authentication protocol. Solid arrows indicate the initial handshake message flows, and dotted arrows indicate subsequent authenticated “keep alive” messages. Note that #1 and #2 in messages 2 and 3 indicate that the contents of messages 1 and 2 are included in the calculation of the HMAC, as is done in TLS 1.2.

2. **(G2) Proof of Liveness:** The Prover and Verifier will be asked to demonstrate that they remain on the call throughout its duration.

Note that we do not aim to achieve voice confidentiality. As discussed in Section 2, the path between two telephony participants is likely to include a range of codec transformations, making the bitwise representation of voice vary significantly between source and destination. Accordingly, end-to-end encryption of voice content is not currently possible given the relatively low channel bitrate and large impact of transcoding. Solutions such as Silent Circle [7] and RedPhone [1] are able to achieve this guarantee strictly because they are VoIP clients that traverse *only* data networks and therefore do not experience transcoding. However, as we discuss in Section 7, our techniques enable the creation of a low-bandwidth channel that can be used to protect the confidentiality and integrity of weak client authentication credentials.

5 AuthLoop Protocol

This section describes the design and implementation of the AuthLoop protocol.

5.1 Design Considerations

Before describing the full protocol, this section briefly discusses the design considerations that led to the AuthLoop authentication protocol. As previously mentioned, we are constrained in that there is no fully-fledged Public Key Infrastructure, meaning that Verifiers (i.e., end users) do not universally possess a strong credential. Moreover, because we are limited to transmission over the audio channel, the AuthLoop protocol must be highly bandwidth efficient.

The most natural choice for AuthLoop would be to reuse an authentication protocol such as Needham-Schroeder [50]. Reusing well-understood security protocols has great value. However, Needham-Schroeder is inappropriate because it assumes that both sides have public/private key pairs or can communicate with a third party for session key establishment. Goal G1 is therefore not practically achievable in real telephony systems if Needham-Schroeder is used. This protocol is also unsuitable as it does not establish session keys, meaning that achieving G2 would require frequent re-execution of the entire authentication protocol, which is likely to be highly inefficient.

TLS can achieve goals G1 and G2, and already does so for a wide range of traditional applications on the Web. Unfortunately, the handshaking and negotiation phases of TLS 1.2 require significant bandwidth. As we demonstrate in Section 3, unmodified use of this protocol can require an average of 97 seconds before authentication can be completed. However, because it can achieve goals G1 and G2, TLS 1.2 is useful as a template for our protocol, and we discuss what could be considered a highly-optimized version below. We note that while TLS 1.3 provides great promise for reducing handshaking costs, the current draft version requires more bandwidth than the AuthLoop protocol.

5.2 Protocol Definition

Figure 5 provides a formal definition for our authentication protocol. We describe this protocol below, and provide details about its implementation and parameterization (e.g., algorithm selection) in Section 5.4.

The AuthLoop protocol begins immediately after a

call is terminated.⁵ Either party, the Prover P (e.g., a call center) or the Verifier V (e.g., the end user) can initiate the call. V then transmits its identity (i.e., phone number) and a nonce N_V to P . Upon receiving this message, P transmits a nonce N_P , its certificate C_P , and signs the contents of the message to bind the nonce to its identity. Its identity, P , is transmitted via Caller ID and is also present in the certificate.

V then generates a pre-master secret S , and uses S to generate a session key k , which is the result of $HMAC(S, N_P, N_V)$. V then extracts P 's public key from the certificate, encrypts S using that key and then computes $HMAC(k, 'VRFY', \#1, \#2)$, where 'VRFY' is a literal string, and #1 and #2 represent the contents of messages 1 and 2. V then sends S and the HMAC to P . P decrypts the pre-master secret and uses it to similarly calculate k , after which it calculates $HMAC(k, 'PROV', \#1, \#2)$, which it then returns to V .

At this time, P has demonstrated knowledge of the private key associated with the public key included in its certificate, thereby authenticating the asserted identity. If the Prover does not provide the correct response, its claim of the Caller ID as its identity is rejected. Security goal G1 is therefore achieved. Moreover, P and V now share a session key k , which can be subsequently used to provide continued and efficient proofs (i.e., HMACs over incrementing nonces) that they remain on the call, thereby achieving Goal G2.

We note that the session key generation step between messages 2 and 3 can be extended to provide keys for protecting confidentiality and integrity (as is done in most TLS sessions). While these keys are not of value for voice communications (given the narrow bitrate of our channel), they can be used to protect client authentication credentials. We discuss this in greater detail in Section 7.

5.3 Formal Verification

We believe that our protocol is secure via inspection. However, to provide stronger guarantees, we use the Proverif v1.93 [22] automatic cryptographic protocol verifier to reason about the security of the AuthLoop handshake. Proverif requires that protocols be rewritten as Horn clauses and modeled in Pi Calculus, from which it can then reason about secrecy and authentication in the Dolev-Yao setting. AuthLoop was represented by a total of 60 lines of code, and Proverif verified the secrecy of the session key k . Further details about configuration will be available in our technical report.

⁵This is the telephony term for "delivered to its intended destination," and signifies the beginning of a call, not its end.

Table 2: Authloop Message Sizes

Message Field	Size(Bits)
Verifier Hello	144
Nonce	96
Cert Ident Number	40
Protocol Command	8
Prover Hello	1692
Nonce	96
Certificate (optional)	1592
Protocol Command	8
Verifier Challenge	1312
Encrypted Premaster Secret	1224
HMAC	80
Protocol Command	8
Prover Response	88
HMAC	80
Protocol Command	8
Total With Certificate	3236
Total Without Certificate	1648

5.4 Implementation Parameters

Table 2 provides accounting of every bit used in the AuthLoop protocol for each message. Given the tight constraints on the channel, we use the following parameters and considerations to implement our protocol as efficiently as possible while still providing strong security guarantees.

We use elliptic curve cryptography for public key primitives. We used the Pyelliptic library for Python [9], which is a Python wrapper around OpenSSL. Keys were generated on curve `sect283r1`, and keys on this curve provide security equivalent to RSA 3456 [56]. For keyed hashes, we use SHA-256 as the underlying hash function for HMACs. To reduce transmission time, we compute the full 256-bit HMAC and truncate the result to 80 bits. Because the security factor of HMAC is dependent almost entirely on the length of the hash, this truncation maintains a security factor of 2^{-80} [21]. This security factor is a commonly accepted safe value [49] for the near future, and as our data transmission improves, the security factor can increase as well.

While similar to TLS 1.2, we have made a few important changes to reduce overhead. For instance, we do not perform cipher suite negotiation in every session and instead assume the default use of AES256-GCM and SHA256. Our link layer header contains a bit field indicating whether negotiation is necessary; however, it is our belief that starting with strong defaults and negotiating in the rare scenario where negotiation is necessary is critical to saving bandwidth for AuthLoop. Similarly, we are able to exclude additional optional information (e.g.,

compression types supported) and the rigid TLS Record format to ensure that our overhead is minimized.

We also limit the contents of certificates. Our certificates consist of a protocol version, the prover’s phone number, claimed identification (i.e., a name), validity period, unique certificate identification number, the certificate owner’s ECC public key and a signature. Because certificate transmission comprises nearly half of the total transmission time, we implemented two variants of AuthLoop: the standard handshake and a version with a verifier-cached certificate. Certificate caching enables a significantly abbreviated handshake. For certificate caching, we include a 16-bit certificate identifier that the verifier sends to the prover to identify which certificate is cached. We discuss how we limit transmitted certificate chain size to a single certificate in Section 7.

Finally, we keep the most security-sensitive parameters as defined in the TLS specification, including recommended sizes for nonces (96 bits).

While our protocol implementation significantly reduces the overhead compared to TLS 1.2 for this application, there is still room for improvement. In particular, the encrypted pre-master secret requires 1224 bits for the 256-bit premaster secret. This expansion is due to the fact that while RSA has a simple primitive for direct encryption of a small value, with ECC one must use a hybrid encryption model called the Integrated Encryption Scheme (IEC), so a key must be shared separately from the encrypted data. Pyelliptic also includes a SHA-256 HMAC of the ECC keyshare and encrypted data to ensure integrity of the message (which is standard practice in IEC). Because the message already includes an HMAC, in future work we plan to save 256 bits (or 15% of the cached certificate handshake) by including the HMAC of the ECC share into the message HMAC.

6 Evaluation

Previous sections established the need for a custom authentication protocol using a voice channel modem to provide end-to-end authentication for telephone calls. In this section, we describe and evaluate our prototype implementation. In particular, we characterize the error performance of the modem across several audio codecs, compute the resulting actual throughput after layer 2 effects are taken into account, and finally measure the end to end timing of complete handshakes.

6.1 Prototype Implementation

Our prototype implementation consists of software implementing the protocol, link layer, and modem running on commodity PCs. While we envision that AuthLoop

Table 3: Bit Error Rates

Codec	Average Bit Error	Std. Dev
G.711	0.0%	0.0%
AMR-NB	0.3%	0.2%
Speex	0.5%	5%

will eventually be a stand-alone embedded device or implemented in telephone hardware/software, a PC served as an ideal prototyping platform to evaluate the system.

We implemented the AuthLoop protocol in Python using the Pyelliptic library for cryptography. We also implemented the link layer in Python. Our modem was written in Matlab, and that code is responsible for modulating data, demodulating data, and sending and receiving samples over the voice channel. We used the Python Engine for Matlab to integrate our modem with Python. Our choice of Matlab facilitated rapid prototyping and development of the modem, but the Matlab runtime placed a considerable load on the PCs running the prototype. Accordingly, computation results, while already acceptable, should improve for embedded implementations.

We evaluate the modem and handshake using software audio channels configured to use one of three audio codecs: G.711 (μ -law), Adaptive MultiRate Narrow Band (AMR-NB), and Speex. These particular codecs were among the most common codecs used for landline audio compression, cellular audio, and VoIP audio, respectively. We use the sox[10] implementations of G.711 and AMR-NB and the ffmpeg[8] implementation of Speex. We use software audio channels to provide a common baseline of comparison, as no VoIP client or cellular device supports all of these codecs.

As link layer performance depends only on the bit error characteristics of the modem, we evaluate the link layer using a software loopback with tunable loss characteristics instead of a voice channel. This allowed us to fully and reproducibly test and evaluate the link layer.

6.2 Modem Evaluation

The most important characteristic of the modem is its resistance to bit errors. To measure bit error, we transmit 100 frames of 2000 random bits⁶ each and measure the bit error after reception.

Table 3 shows the average and standard deviation of the bit error for various codecs. The modem saw no bit errors on the G.711 channel; this is reflective of the fact that G.711 is high-quality channel with very minimal processing and compression. AMR-NB and Speex

⁶2000 bits was chosen as the first “round” number larger than the largest message in the AuthLoop handshake.

Table 4: Link Layer Transmission of 2000 bits

Bit Error Rate	Transmission Time	Goodput
0.1%	4.086 s (0.004)	490 bps
1%	6.130 s (0.009)	326 bps
2%	11.652 s (0.007)	172 bps

both saw minimal bit error as well, though Speex had a much higher variance in errors. Speex had such a high variance because one frame was truncated, resulting in a higher average error despite the fact the other 99 frames were received *with no error*.

6.3 Link Layer Evaluation

The most important characteristic of the link layer is its ability to optimize goodput – the actual amount of application data transmitted per unit time (removing overhead from consideration).

Table 4 shows as a function of bit error the transmission time and the goodput of the protocol compared to the theoretical optimal transmission time and goodput. The optimal numbers are computed from the optimal bit time (at 500 bits per second) plus 40ms of header and footer. The experimental numbers are the average of transmission of 50 messages with 2000 bits each. The table shows that in spite of high bit error rates (up to 2%) the link layer is able to complete message transmission. Of course, the effect of bit errors on goodput is substantial at larger rates. Fortunately, low bit error rates (e.g. 0.1%) result in a minor penalty to goodput – only 5bps lower than the optimal rate. Higher rates have a more severe impact, resulting in 65.8% and 34.7% of optimal goodput for 1% and 2% loss. Given our observations of bit error rates at less than 0.5% for all codecs, these results demonstrate that our Link Layer retransmission parameters are set with an acceptable range.

6.4 Handshake Evaluation

To evaluate the complete handshake, we measure the complete time from handshake start to handshake completion from the verifier’s perspective. We evaluate both variants of the handshake: with and without the prover sending a certificate. Handshakes requiring a certificate exchange will take much longer than handshakes without a certificate. This is a natural consequence of simply sending more data.

Table 5 shows the total handshake times for calls over each of the three codecs. These results are over 10 calls each. Note that these times are corrected to remove the effects of instrumentation delays and artificial delays caused by IPC among the different components of our

prototype that would be removed or consolidated in deployment.

From the verifier perspective, we find that cached-certificate exchanges are quite fast – averaging 4.844 seconds across all codecs. When certificates are not cached, our overall average time is 8.977 seconds. Differences in times taken for certificate exchanges for different codecs are caused by the relative underlying bit error rate of each codec. G.711 and Speex have much lower error rates than AMR-NB, and this results in a lower overall handshake time. In fact, because those codecs saw no errors during the tests, their execution times were virtually identical.

Most of the time spent in the handshake is spent in transmitting messages over the voice channel. In fact, transmission time accounts for 99% of our handshake time. Computation and miscellaneous overhead average to less than 50 milliseconds for all messages. This indicates that AuthLoop is computationally minimal and can be implemented on a variety of platforms.

7 Discussion

This section provides a discussion of client authentication, public key infrastructure, and deployment considerations for AuthLoop.

7.1 Client Credentials

Up until this point, we have focused our discussion around strong authentication of one party in the phone call (i.e., the Prover). However, clients already engage in a weaker “application-layer” authentication when talking to many call centers. For instance, when calling a financial institution or ISP, users enter their account number and additional values including PINs and social security numbers. Without one final step, our threat model would allow for an adversary to successfully steal such credentials as follows: An adversary would launch a 3-Way call to both the victim client and the targeted institution. After passively observing the successful handshake, the adversary could capture the client’s credentials (i.e., DTMF tone inputs) and hang up both ends of the call. The adversary could then call the targeted institution back spoofing the victim’s Caller ID and present the correct credentials.

One of the advantages of TLS is that it allows for the generation of multiple session keys, for use not only in continued authentication, but also in the protection of data confidentiality and integrity. AuthLoop is no different. While the data throughput enabled by our modem is low, it is sufficiently large enough to carry encrypted copies of client credentials. Accordingly, an adversary attempting to execute the above attack would be unable to do so successfully because this sensitive information

Table 5: Handshake completion times

Codec	Cached Certificate	Certificate Exchanged
G.711	4.463 s (0.000)	8.279 s (0.000)
AMR-NB	5.608 s (0.776)	10.374 s (0.569)
Speex	4.427 s (0.000)	8.279 s (0.000)
Average	4.844 s	8.977 s

could easily be passed through AuthLoop (and therefore useless in a second session). Moreover, because users are already accustomed to entering such information when interacting with these entities, the user experience could continue without any observable difference.

7.2 Telephony PKI

One of the most significant problems facing SSL/TLS is its trust model. X.509 certificates are issued by a vast number of Certificate Authorities (CAs), whose root certificates can be used to verify the authenticity of a presented certificate. Unfortunately, the unregulated nature of who can issue certificates to whom (i.e., what authority does X have to verify and bind names to entity Y ?) and even who can act as a CA have been known since the inception of the current Public Key Infrastructure [37]. This weakness has lead to a wide range of attacks, and enabled both the mistaken identity of domain owners and confusion as to which root-signed certificate can be trusted. Traditional certificates present another challenge in this environment - the existence of long verification chains in the presence of the bitrate limited audio channel means that the blind adoption of the Internet's traditional PKI model will simply fail if applied to telephony systems. As we demonstrated in our experiment in Table 1, transmitting the entirety of long certificate chains would simply be detrimental to the performance of AuthLoop.

The structure of telephony networks leads to a natural, single rooted PKI system. Competitive Local Exchange Carriers (CLECs) are assigned blocks of phone numbers by the North American Numbering Plan Association (NANPA), and ownership of these blocks is easily confirmed through publicly posted resources such as NPA/NXX databases in North America. A similar observation was recently made in the secure Internet routing community, and resulted in the proposal of the Resource Public Key Infrastructure (RPKI) [45]. The advantage to this approach is that because all allocation of phone numbers is conducted under the ultimate authority of NANPA, all valid signatures on phone numbers must ultimately be rooted in a NANPA certificate. This Telephony Public Key Infrastructure (TPKI) reduces the length of certificate chains and allows us to easily store the root and all CLEC certificates in the US and asso-

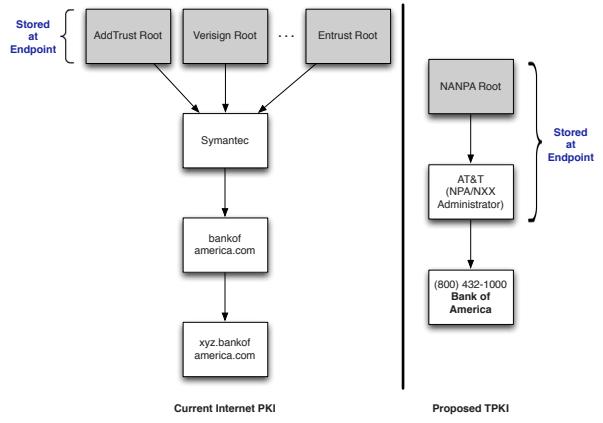


Figure 6: The Telephony Public Key Infrastructure (TPKI). Unlike in Internet model, the TPKI has a single root (NANPA) which is responsible for all block allocation, and a limited second level of CLECs who administer specific numbers. Accordingly, only the certificate for the number claimed in the current call needs to be sent during the handshake.

ciated territories (≈ 700 [46]) in just over 100 KiB of storage (1600 bits per certificate \times 700). Alternatively, if certificates are only needed for toll free numbers, a single certificate for the company that administers all such numbers (i.e., Somos, Inc.) would be sufficient.

Figure 6 shows the advantages of our approach. Communicating with a specific server (xyz.bankofamerica.com) may require the transmission of three or more certificates before identity can be verified. Additionally, the existence of different roots adds confusion to the legitimacy of any claimed identity. Our proposed TPKI relies on a single NANPA root, and takes advantage of the relatively small total number of CLECs to require only single certificate for the calling number to be transmitted during the handshake. We leave further discussion of the details of the proposed TPKI (e.g., revocation, etc) to our future work.

7.3 Deployment Considerations

As our experiments demonstrate that AuthLoop is bandwidth and not processor bound, we believe that these

techniques can be deployed successfully across a wide range of systems. For instance, AuthLoop can be embedded directly into new handset hardware. Moreover, it can be used immediately with legacy equipment through external adapters (e.g., Raspberry Pi). Alternatively, AuthLoop could be loaded onto mobile devices through a software update to the dialer, enabling large numbers of devices to immediately benefit.

Full deployments have the opportunity to make audio signaling of AuthLoop almost invisible to the user. If AuthLoop is in-line with the call audio, the system can remove AuthLoop transmissions from the audio sent to the user. In other words, users will never hear the AuthLoop handshakes or keep-alive messages. While our current strategy is to minimize the volume of the signaling so as to not interrupt a conversation (as has been done in other signaling research [62]), we believe that the in-line approach will ultimately provide the greatest stability and least intrusive user experience.

Lastly, we note that because AuthLoop is targeted across all telephony platforms, a range of security indicators will be necessary for successfully communicating authenticated identity to the user. However, given the limitations of space and the breadth of devices and their interfaces, we leave this significant exploration to our future work.

8 Conclusions

Phone systems serve as the trusted carriers of some of our most sensitive communications. In spite of this trust, authentication between two end points across this heterogeneous landscape was previously not possible. In this paper, we present AuthLoop to address this challenge. We began by designing a modem and supporting link layer protocol for the reliable delivery of data over a voice channel. With the limitations of this channel understood, we then presented a security model and protocol to provide explicit authentication of an assertion of Caller ID, and discussed ways in which client credentials could be subsequently protected. Finally, we demonstrated that AuthLoop reduced execution time by over an order of magnitude on average when compared to the direct application of TLS 1.2 to this problem. In so doing, we have demonstrated that end-to-end authentication is indeed possible across modern telephony networks.

Acknowledgment

The authors would like to thank our anonymous reviewers for their helpful comments and colleagues at the Florida Institute for Cybersecurity Research for their assistance in preparing this manuscript.

This work was supported in part by the US National Science Foundation under grant numbers CNS-1617474, CNS-1526718 and CNS-1464088. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- [1] RedPhone. <https://play.google.com/store/apps/details?id=org.thoughtcrime.redphone>.
- [2] Directory of Unknown Callers. <http://www.800notes.com/>, 2015.
- [3] GSMK CryptoPhone. <http://www.cryptophone.de/en/>, 2015.
- [4] Nomorobo. <https://www.nomorobo.com/>, 2015.
- [5] PGPfone - Pretty Good Privacy Phone. <http://www.pgpi.org/products/pgpfone/>, 2015.
- [6] Signal. <https://itunes.apple.com/us/app/signal-private-messenger/id874139669?mt=8>, 2015.
- [7] Silent Circle. <https://www.silentcircle.com/>, 2015.
- [8] ffmpeg. <https://www.ffmpeg.org>, 2016.
- [9] Pyelliptic. <https://pypi.python.org/pypi/pyelliptic>, 2016.
- [10] sox. <http://sox.sourceforge.net/Main/HomePage>, 2016.
- [11] 3rd Generation Partnership Project. A Guide to 3rd Generation Security. Technical Report 33.900 version 1.2.0, 2000.
- [12] 3rd Generation Partnership Project. 3G Security Principles and Objectives (3GPP TS 33.120). 2001.
- [13] 3rd Generation Partnership Project. 3GPP TS 23.228 IP Multimedia Subsystem (IMS). (Release 11), 2012.
- [14] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer. Here's my cert, so trust me, maybe? Understanding TLS errors on the web. In *Proceedings of the 22nd International Conference on World Wide Web (WWW)*, pages 59–70, 2013.

- [15] D. Akhawe and A. P. Felt. Alice in Warningland: A large-scale field study of browser security warning effectiveness. In *Proceedings of the USENIX Security Symposium*, 2013.
- [16] F. Alegre, G. Soldi, and N. Evans. Evasion and obfuscation in automatic speaker verification. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 749–753, 2014.
- [17] F. Alegre and R. Vipperla. On the vulnerability of automatic speaker recognition to spoofing attacks with artificial signals. In *Proceedings of the 20th European Signal Processing Conference (EUSIPCO)*, pages 36–40, 2012.
- [18] V. Balasubramaniyan, A. Poonawalla, M. Ahamad, M. Hunter, and P. Traynor. PinDr0p: Using Single-Ended Audio Features to Determine Call Provenance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.
- [19] E. Barkan, E. Biham, and N. Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. *Journal of Cryptology*, 21(3):392–429, 2008.
- [20] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, and K. R. Butler. Forced perspectives: Evaluating an SSL trust enhancement at scale. In *Proceedings of the 2014 Internet Measurement Conference (IMC)*, pages 503–510. ACM, 2014.
- [21] M. Bellare. New Proofs for NMAC and HMAC Security without Collision-Resistance. *Advances in Cryptology - CRYPTO '06*, 2006.
- [22] B. Blanchet. ProVerif: Cryptographic protocol verifier in the formal model. <http://www.proverif.ens.fr/>, 2016.
- [23] H. K. Bokharaei, A. Sahraei, Y. Ganjali, R. Kerlapura, and A. Nucci. You can SPIT, but you can't hide: Spammer identification in telephony networks. In *Proceedings of the IEEE INFOCOM*, pages 41–45, 2011.
- [24] R. Bresciani. The ZRTP protocol analysis on the Diffie-Hellman mode. *Foundations and Methods Research Group*, 2009.
- [25] R. Bresciani, S. Superiore, S. Anna, and I. Pisa. The ZRTP protocol security considerations. Technical Report LSV-07-20, 2007.
- [26] Y. J. Choi and S. J. Kim. An improvement on privacy and authentication in GSM. In *Proceedings of Workshop on Information Security Applications (WISA)*, 2004.
- [27] J. Clark and P. C. Van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 511–525, 2013.
- [28] Communications Fraud Control Association (CFCA). 2013 Global Fraud Loss Survey. http://www.cvidya.com/media/62059/global-fraud-loss_survey2013.pdf, 2013.
- [29] I. Dacosta, M. Ahamad, and P. Traynor. Trust No One Else: Detecting MITM Attacks Against SS-L/TLS Without Third-Parties. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2012.
- [30] I. Dacosta, V. Balasubramaniyan, M. Ahamad, and P. Traynor. Improving Authentication Performance of Distributed SIP Proxies. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 22(11):1804–1812, 2011.
- [31] I. Dacosta and P. Traynor. Proxychain: Developing a Robust and Efficient Authentication Infrastructure for Carrier-Scale VoIP Networks. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.
- [32] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI)*, CHI '06, New York, NY, USA, 2006. ACM.
- [33] A. Dhananjay, A. Sharma, M. Paik, J. Chen, T. K. Kuppusamy, J. Li, and L. Subramanian. Hermes: Data transmission over unknown voice channels. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking, MobiCom*, New York, NY, USA, 2010. ACM.
- [34] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC)*, pages 475–488, New York, NY, USA, 2014. ACM.
- [35] S. Egelman, L. F. Cranor, and J. Hong. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2008.

- [36] C. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. IETF, RFC 2693, 1999.
- [37] C. Ellison and B. Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.
- [38] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The SSL landscape: a thorough analysis of the x.509 PKI using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet Measurement Conference (IMC)*, pages 427–444, 2011.
- [39] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing forged SSL certificates in the wild. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2014.
- [40] N. Jiang, Y. Jin, A. Skudlark, W.-L. Hsu, G. Jacobson, S. Prakasam, and Z.-L. Zhang. Isolating and analyzing fraud activities in a large cellular network via voice call graph analysis. In *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys)*, page 253, 2012.
- [41] Q. Jin, A. R. Toth, A. W. Black, and T. Schultz. Is voice transformation a threat to speaker identification? In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4845–4848. IEEE, 2008.
- [42] N. N. Katugampala, K. T. Al-Naimi, S. Villette, and A. M. Kondoz. Real-time end-to-end secure voice communications over GSM voice channel. *2005 European Signal Processing Conference*, pages 1–4, 2005.
- [43] P. Koopman and T. Chakravarty. Cyclic redundancy code (CRC) polynomial selection for embedded networks. In *2004 International Conference on Dependable Systems and Networks*, pages 145–154, June 2004.
- [44] C. Lee, M. Hwang, and W. Yang. Enhanced privacy and authentication for the global system for mobile communications. *Wireless Networks*, 5(4):231–243, 1999.
- [45] M. Lepinski, R. Barnes, and S. Kent. An Infrastructure to Support Secure Internet Routing. IETF, RFC 6480, 2012.
- [46] Local Search Association. CLEC Information. <http://www.thelsa.org/main/clecinformation.aspx>, 2016.
- [47] B. Mathieu, S. Niccolini, and D. Sisalem. SDRS: A Voice-over-IP spam detection and reaction system. *IEEE Security & Privacy Magazine*, 6(6):52–59, nov 2008.
- [48] B. Moeller and A. Langley. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. Internet-draft, Internet Engineering Task Force, 2014.
- [49] National Institute of Standards and Technology. NIST Special Publication 800-107 Revision 1: Recommendation for Applications Using Approved Hash Algorithms. <http://csrc.nist.gov/publications/nistpubs/800-107-rev1/sp800-107-rev1.pdf>, 2008.
- [50] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
- [51] M. A. Ozkan, B. Ors, and G. Saldamli. Secure voice communication via GSM network. *2011 7th International Conference on Electrical and Electronics Engineering (ELECO)*, pages II–288–II–292, 2011.
- [52] M. Petraschek, T. Hoher, O. Jung, H. Hlavacs, and W. Gansterer. Security and usability aspects of Man-in-the-Middle attacks on ZRTP. *Journal of Universal Computer Science*, 14(5):673–692, 2008.
- [53] A. Ramirez. Theft through cellular ‘clone’ calls. <http://www.nytimes.com/1992/04/07/business/theft-through-cellular-clone-calls.html>, April 7, 1992.
- [54] B. Reaves, E. Shernan, A. Bates, H. Carter, and P. Traynor. Boxed Out: Blocking Cellular Interconnect Bypass Fraud at the Network Edge. In *Proceedings of the USENIX Security Symposium (SECURITY)*, 2015.
- [55] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.
- [56] C. Research. SEC 2: Recommended Elliptic Curve Domain Parameters, January 2010.
- [57] R. Rivest and B. Lampson. SDSI: A Simple Distributed Security Infrastructure. <http://research.microsoft.com/en-us/um/people/blampson/59-sdsi/webpage.html>, 1996.

- [58] S. Rosset, U. Murad, E. Neumann, Y. Idan, and G. Pinkas. Discovery of fraud rules for telecommunications-challenges and solutions. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 409–413, New York, NY, USA, 1999.
- [59] D. Samfat, R. Molva, and N. Asokan. Untraceability in mobile networks. In *Proceedings of the First Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 26–36, 1995.
- [60] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor’s new security indicators. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2007.
- [61] H. Sengar. VoIP Fraud : Identifying a wolf in sheep’s clothing. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 334–345, 2014.
- [62] M. Sherr, E. Cronin, S. Clark, and M. Blaze. Signaling vulnerabilities in wiretapping systems. *IEEE Security & Privacy Magazine*, 3(6):13–25, November 2005.
- [63] M. Shirvanian and N. Saxena. Wiretapping via mimicry: Short voice imitation man-in-the-middle attacks on crypto phones. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 868 – 879.
- [64] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Prentice Hall, Upper Saddle River, N.J, second edition, Jan. 2001.
- [65] J. Sobey, R. Biddle, P. van Oorschot, and A. S. Patrick. Exploring user reactions to new browser cues for extended validation certificates. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2008.
- [66] Y. Stylianou. Voice transformation: A survey. In *Proceedings of the IEEE Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2009.
- [67] TelTech. SpoofCard. <http://www.spoofcard.com/>, 2015.
- [68] M. Toorani and A. Beheshti. Solutions to the GSM security weaknesses. In *Proceedings of the Second International Conference on Next Generation Mobile Applications, Services, and Technologies (NG-MAST)*, pages 576–581, 2008.
- [69] P. Traynor, P. McDaniel, and T. La Porta. *Security for Telecommunications Networks*. Number 978-0-387-72441-6 in Advances in Information Security Series. Springer, August 2008.
- [70] A. Tyrberg. *Data Transmission over Speech Coded Voice Channels*. Master’s Thesis, Linkoping University, 2006.
- [71] Z. Wu, A. Khodabakhsh, C. Demiroglu, J. Yamagishi, D. Saito, T. Toda, and S. King. SAS: A speaker verification spoofing database containing diverse attacks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4440–4444, 2015.
- [72] Z. Wu and H. Li. Voice conversion and spoofing attack on speaker verification systems. In *Proceedings of the Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*. IEEE, 2013.
- [73] P. Zimmermann. Zfone. <http://zfoneproject.com/>, 2015.
- [74] P. Zimmermann and A. Johnston. ZRTP: Media Path Key Agreement for Unicast Secure RTP. IETF, RFC 6189, 2011.
- [75] T. Zoller. TLS & SSLv3 Renegotiation Vulnerability. <http://www.g-sec.lu/practicaltls.pdf>, 2009.

You are Who You Know and How You Behave: Attribute Inference Attacks via Users' Social Friends and Behaviors

Neil Zhenqiang Gong

ECE Department, Iowa State University

neilgong@iastate.edu

Bin Liu

MSIS Department, Rutgers University

BenBinLiu@gmail.com

Abstract

We propose new privacy attacks to infer attributes (e.g., locations, occupations, and interests) of online social network users. Our attacks leverage seemingly innocent user information that is publicly available in online social networks to infer missing attributes of targeted users. Given the increasing availability of (seemingly innocent) user information online, our results have serious implications for Internet privacy – private attributes can be inferred from users' publicly available data unless we take steps to protect users from such inference attacks.

To infer attributes of a targeted user, existing inference attacks leverage either the user's publicly available social friends or the user's behavioral records (e.g., the webpages that the user has liked on Facebook, the apps that the user has reviewed on Google Play), but not both. As we will show, such inference attacks achieve limited success rates. However, the problem becomes *qualitatively* different if we consider both social friends and behavioral records. To address this challenge, we develop a novel model to integrate social friends and behavioral records and design new attacks based on our model. We theoretically and experimentally demonstrate the effectiveness of our attacks. For instance, we observe that, in a real-world large-scale dataset with 1.1 million users, our attack can correctly infer the *cities a user lived in* for 57% of the users; via *confidence estimation*, we are able to increase the attack success rate to over 90% if the attacker selectively attacks a half of the users. Moreover, we show that our attack can correctly infer attributes for significantly more users than previous attacks.

1 Introduction

Online social networks (e.g., Facebook, Google+, and Twitter) have become increasingly important platforms for users to interact with each other, process information, and diffuse social influence. A user in an online social

network essentially has a list of social friends, a digital record of behaviors, and a profile. For instance, behavioral records could be a list of pages liked or shared by the user on Facebook, or they could be a set of mobile apps liked or rated by the user in Google+ or Google Play. A profile introduces the user's self-declared attributes such as majors, employers, and cities lived. To address users' privacy concerns, online social network operators provide users with fine-grained privacy settings, e.g., a user could limit some attributes to be accessible only to his/her friends. Moreover, a user could also create an account without providing any attribute information. *As a result, an online social network is a mixture of both public and private user information.*

One privacy attack of increasing interest revolves around these user attributes [18, 27, 46, 39, 15, 33, 11, 26, 42, 29, 8, 25, 20, 21, 23]. In this *attribute inference attack*, an attacker aims to propagate attribute information of social network users with publicly visible attributes to users with missing or incomplete attribute data. Specifically, the attacker could be any party (e.g., cyber criminal, online social network provider, advertiser, data broker, and surveillance agency) who has interests in users' private attributes. To perform such privacy attacks, the attacker only needs to collect publicly available data from online social networks. Apart from privacy risks, the inferred user attributes can also be used to perform various security-sensitive activities such as spear phishing [37] and attacking personal information based backup authentication [17]. Moreover, an attacker can leverage the inferred attributes to link online users across multiple sites [4, 14, 2, 13] or with offline records (e.g., publicly available voter registration records) [38, 32] to form composite user profiles, resulting in even bigger security and privacy risks.

Existing attribute inference attacks can be roughly classified into two categories, *friend-based* [18, 27, 46, 39, 15, 33, 11, 26, 20, 21, 23] and *behavior-based* [42, 29, 8, 25]. Friend-based attacks are based on the intu-

ition of *you are who you know*. Specifically, they aim to infer attributes for a user using the publicly available user attributes of the user’s friends (or all other users in the social network) and the social structure among them. The foundation of friend-based attacks is *homophily*, meaning that two linked users share similar attributes [30]. For instance, if more than half of friends of a user major in Computer Science at a certain university, the user might also major in Computer Science at the same university with a high probability. Behavior-based attacks infer attributes for a user based on the public attributes of users that are similar to him/her, and the similarities between users are identified by using their behavioral data. The intuition behind behavior-based attacks is *you are how you behave*. In particular, users with the same attributes have similar interests, characteristics, and cultures so that they have similar behaviors. For instance, if a user liked apps, books, and music tracks on Google Play that are similar to those liked by users originally from China, the user might also be from China. Likewise, previous measurement study [43] found that some apps are only popular in certain cities, implying the possibility of inferring cities a user lived in using the apps the user used or liked.

However, these inference attacks consider either social friendship structures or user behaviors, but not both, and thus they achieve limited inference accuracy as we will show in our experiments. Moreover, the problem of inferring user attributes becomes qualitatively different if we consider both social structures and user behaviors because features derived from them differ from each other, show different sparsity, and are at different scales. We show in our evaluation that simply concatenating features from the two sources of information regresses the overall results and reduces attack success rates.

Our work: In this work, we aim to combine social structures and user behaviors to infer user attributes. To this end, we first propose a *social-behavior-attribute (SBA)* network model to gracefully integrate social structures, user behaviors, and user attributes in a unified framework. Specifically, we add additional nodes to a social structure, each of which represents an attribute or a behavior; a link between a user and an attribute node represents that the user has the corresponding attribute, and that a user has a behavior is encoded by a link between the user and the corresponding behavior node.

Second, we design a *vote distribution attack (VIAL)* under the SBA model to perform attribute inference. Specifically, VIAL iteratively distributes a fixed vote capacity from a *targeted user* whose attributes we want to infer to all other users in the SBA network. A user receives a high vote capacity if the user and the targeted user are structurally similar in the SBA network, e.g., they have similar social structures and/or have performed similar behaviors. Then, each user votes for its attributes

via dividing its vote capacity to them. We predict the target user to own attributes that receive the highest votes.

Third, we evaluate VIAL both theoretically and empirically; and we extensively compare VIAL with several previous attacks for inferring majors, employers, and locations using a large-scale dataset with 1.1 million users collected from Google+ and Google Play. For instance, we observe that our attack can correctly infer the cities a user lived in for 57% of the users; via *confidence estimation*, we are able to increase the success rate to over 90% if the attacker selectively attacks a half of the users. Moreover, we find that our attack VIAL substantially outperforms previous attacks. Specifically, for Precision, VIAL improves upon friend-based attacks and behavior-based attacks by over 20% and around 100%, respectively. These results imply that an attacker can use our attack to successfully infer private attributes of substantially more users than previous attacks.

In summary, our key contributions are as follows:

- We propose the *social-behavior-attribute (SBA)* network model to integrate social structures, user behaviors, and user attributes.
- We design the *vote distribution attack (VIAL)* under the SBA model to perform attribute inference.
- We demonstrate the effectiveness of VIAL both theoretically and empirically. Moreover, we observe that VIAL correctly infers attributes for substantially more users than previous attacks via evaluations on a large-scale dataset collected from Google+ and Google Play.

2 Problem Definition and Threat Model

Attackers: The attacker could be any party who has interests in user attributes. For instance, the attacker could be a cyber criminal, online social network provider, advertiser, data broker, or surveillance agency. Cyber criminals can leverage user attributes to perform targeted social engineering attacks (now often referred to as spear phishing attacks [37]) and attacking personal information based backup authentication [17]; online social network providers and advertisers could use the user attributes for targeted advertisements; data brokers make profit via selling the user attribute information to other parties such as advertisers, banking companies, and insurance industries [1]; and surveillance agency can use the attributes to identify users and monitor their activities.

Collecting publicly available social structures and behaviors: To perform attribute inference attacks, an attacker first needs to collect publicly available information. In particular, in our attacks, an attacker needs to

collect social structures, user profiles, and user behaviors from online social networks. Such information can be collected via writing web crawlers or leveraging APIs developed by the service providers. Next, we formally describe these publicly available information.

We use an undirected¹ graph $G_s = (V_s, E_s)$ to represent a social structure, where edges in E_s represent social relationships between the nodes in V_s . We denote by $\Gamma_{u,S} = \{v | (u, v) \in E_s\}$ as the set of social neighbors of u . In addition to social network structure, we have behaviors and categorical attributes for nodes. For instance, in our Google+ and Google Play dataset, nodes are Google+ users, and edges represent friendship between users; behaviors include the set of items (e.g., apps, books, and movies) that users rated or liked on Google Play; and node attributes are derived from user profile information and include fields such as major, employer, and cities lived.

We use binary representation for user behaviors. Specifically, we treat various objects (e.g., the Android app “Angry Birds”, the movie “The Lord of the Rings”, and the webpage “facebook.com”) as binary variables, and we denote by m_b the total number of objects. Behaviors of a node u are then represented as a m_b -dimensional binary column vector \vec{b}_u with the i th entry equal to 1 when u has performed a certain action on the i th object (*positive behavior*) and -1 when u does not perform the action on it (*negative behavior*). For instance, when we consider user review behaviors for Google+ users, objects could be items such as apps, books, and movies available in Google Play, and the action is *review*; 1 represents that the user reviewed the corresponding item and -1 means the opposite. For Facebook users, objects could be webpages; 1 represents that the user liked or shared the corresponding webpage and -1 means that the user did not. We denote by $\mathbf{B} = [\vec{b}_1 \vec{b}_2 \cdots \vec{b}_{n_s}]$ the behavior matrix for all nodes.

We distinguish between *attributes* and *attribute values*. For instance, major, employer, and location are different attributes; and each attribute could have multiple attribute values, e.g., major could be Computer Science, Biology, or Physics. A user might own a few attribute values for a single attribute. For example, a user that studies Physics for undergraduate education but chooses to pursue a Ph.D. degree in Computer Science has two values for the attribute major. Again, we use a binary representation for each attribute value, and we denote the number of distinct attribute values as m_a . Then attribute information of a node u is represented as a m_a -dimensional binary column vector \vec{a}_u with the i th entry equal to 1 when u has the i th attribute value (*positive attribute*) and -1 when u does not have it (*negative attribute*).

¹Our attacks can also be generalized to directed graphs.

tribute). We denote by $\mathbf{A} = [\vec{a}_1 \vec{a}_2 \cdots \vec{a}_{n_s}]$ the attribute matrix for all nodes.

Attribute inference attacks: Roughly speaking, an attribute inference attack is to infer the attributes of a set of targeted users using the collected publicly available information. Formally, we define an attribute inference attack as follows:

Definition 1 (Attribute Inference Attack). *Suppose we are given $T = (G_s, \mathbf{A}, \mathbf{B})$, which is a snapshot of a social network G_s with a behavior matrix \mathbf{B} and an attribute matrix \mathbf{A} , and a list of targeted users V_t with social friends $\Gamma_{v,S}$ and binary behavior vectors \vec{b}_v for all $v \in V_t$, the attribute inference attack is to infer the attribute vectors \vec{a}_v for all $v \in V_t$.*

We note that a user setting the friend list to be private could also be vulnerable to inference attacks. This is because the user’s friends could set their friend lists publicly available. The attacker can collect a social relationship between two users if at least one of them sets the friend list to be public. Moreover, we assume the users and the service providers are not taking other steps (e.g., obfuscating social friends [19] or behaviors [42, 9]) to defend against inference attacks.

Applying inferred attributes to link users across multiple online social networks and with offline records: We stress that an attacker could leverage our attribute inference attacks to further perform other attacks. For instance, a user might provide different attributes on different online social networks. Thus, an attacker could combine user attributes across multiple online social networks to better profile users, and an attacker could leverage the inferred user attributes to do so [4, 14, 2, 13]. Moreover, an attacker can further use the inferred user attributes to link online users with offline records (e.g., voter registration records) [38, 32], which results in even bigger security and privacy risks, e.g., more sophisticated social engineering attacks. We note that even if the inferred user attributes (e.g., major, employer) seem not private for some targeted users, an attacker could use them to link users across multiple online sites and with offline records.

3 Social-Behavior-Attribute Framework

We describe our *social-behavior-attribute (SBA)* network model, which integrates social structures, user behaviors, and user attributes in a unified framework. To perform our inference attacks, an attacker needs to construct a SBA network from his/her collected publicly available social structures, user attributes, and behaviors.

Given a social network $G_s = (V_s, E_s)$ with m_b behavior objects, a behavior matrix \mathbf{B} , m_a distinct attribute values, and an attribute matrix \mathbf{A} , we create an augmented

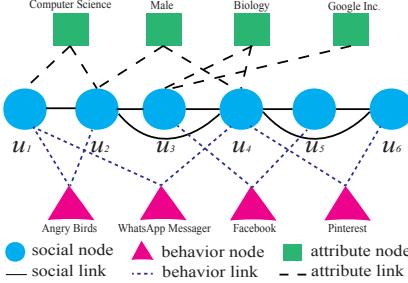


Figure 1: Social-behavior-attribute network.

network by adding m_b additional nodes to G_s , with each node corresponding to a behavior object, and another m_a additional nodes to G_s , with each additional node corresponding to an attribute value. For each node u in G_s with positive attribute a or positive behavior b , we create an undirected link between u and the additional node corresponding to a or b in the augmented network. Moreover, we add the targeted users into the augmented network by connecting them to their friends and the additional nodes corresponding to their positive behaviors. We call this augmented network *social-behavior-attribute (SBA)* network since it integrates the interactions among social structures, user behaviors, and user attributes.

Nodes in the SBA framework corresponding to nodes in G_s or targeted users in V_t are called *social nodes*, nodes representing behavior objects are called *behavior nodes*, and nodes representing attribute values are called *attribute nodes*. Moreover, we use S , B , and A to represent the three types of nodes, respectively. Links between social nodes are called *social links*, links between social nodes and behavior nodes are called *behavior links*, and links between social nodes and attribute nodes are called *attribute links*. Note that there are no links between behavior nodes and attribute nodes. Fig. 1 illustrates an example SBA network, in which the two social nodes u_5 and u_6 correspond to two targeted users. The behavior nodes in this example correspond to Android apps, and a behavior link represents that the corresponding user used the corresponding app. Intuitively, the SBA framework explicitly describes the sharing of behaviors and attributes across social nodes.

We also place weights on various links in the SBA framework. These link weights balance the influence of social links versus behavior links versus attribute links.² For instance, weights on social links could represent the tie strengths between social nodes. Users with stronger tie strengths could be more likely to share the same attribute values. The weight on a behavior link could in-

dicate the predictiveness of the behavior in terms of the user’s attributes. In other words, a behavior link with a higher weight means that performing the corresponding behavior better predicts the attributes of the user. For instance, if we want to predict user gender, the weight of the link between a female user and a mobile app tracking women’s monthly periods could be larger than the weight of the link between a male user and the app. Weights on attribute links can represent the degree of affinity between users and attribute values. For instance, an attribute link connecting the user’s hometown could have a higher weight than the attribute link connecting a city where the user once travelled. We discuss how link weights can be learnt via machine learning in Section 8.

We denote a SBA network as $G = (V, E, w, t)$, where V is the set of nodes, $n = |V|$ is the total number of nodes, E is the set of links, $m = |E|$ is the total number of links, w is a function that maps a link to its link weight, i.e., w_{uv} is the weight of link (u, v) , and t a function that maps a node to its node type, i.e., t_u is the node type of u . For instance, $t_u = S$ means that u is a social node. Additionally, for a given node u in the SBA network, we denote by Γ_u , $\Gamma_{u,S}$, $\Gamma_{u,B}$, and $\Gamma_{u,A}$ respectively the sets of *all neighbors*, *social neighbors*, *behavior neighbors*, and *attribute neighbors* of u . Moreover, for links that are incident from u , we use d_u , $d_{u,S}$, $d_{u,B}$, and $d_{u,A}$ to denote the sum of weights of all links, weights of links connecting social neighbors, weights of links connecting behavior neighbors, and weights of links connecting attribute neighbors, respectively. More specifically, we have $d_u = \sum_{v \in \Gamma_u} w_{uv}$ and $d_{u,Y} = \sum_{v \in \Gamma_{u,Y}} w_{uv}$, where $Y = S, B, A$.

Furthermore, we define two types of *hop-2 social neighbors* of a social node u , which share common behavior neighbors or attribute neighbors with u . In particular, a social node v is called a *behavior-sharing social neighbor* of u if v and u share at least one common behavior neighbor. For instance, in Fig. 1, both u_2 and u_4 are behavior-sharing social neighbors of u_1 . We denote the set of behavior-sharing social neighbors of u as $\Gamma_{u,BS}$. Similarly, we denote the set of attribute-sharing social neighbors of u as $\Gamma_{u,AS}$. Formally, we have $\Gamma_{u,BS} = \{v | t(v) = S \& \Gamma_{v,B} \cap \Gamma_{u,B} \neq \emptyset\}$ and $\Gamma_{u,AS} = \{v | t(v) = S \& \Gamma_{v,A} \cap \Gamma_{u,A} \neq \emptyset\}$. We note that our definitions of $\Gamma_{u,BS}$ and $\Gamma_{u,AS}$ also include the social node u itself. These notations will be useful in describing our attack.

4 Vote Distribution Attack (VIAL)

4.1 Overview

Suppose we are given a SBA network G which also includes the social structures and behaviors of the targeted users, our goal is to infer attributes for every targeted user. Specifically, for each targeted user v , we compute

²In principle, we could also assign weights to nodes to incorporate their relative importance. However, our attack does not rely on node weights, so we do not discuss them.

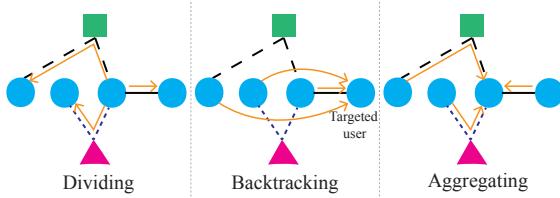


Figure 2: Illustration of our three local rules.

the similarity between v and each attribute value, and then we predict that v owns the attribute values that have the highest similarity scores. In a high-level abstraction, VIAL works in two phases.

- **Phase I.** VIAL iteratively distributes a fixed vote capacity from the targeted user v to the rest of users in Phase I. The intuitions are that a user receives a high vote capacity if the user and the targeted user are structurally similar in the SBA network (e.g., share common friends and behaviors), and that the targeted user is more likely to have the attribute values belonging to users with higher vote capacities. After Phase I, we obtain a vote capacity vector \vec{s}_v , where \vec{s}_{vu} is the vote capacity of user u .
- **Phase II.** Intuitively, if a user with a certain vote capacity has more attribute values, then, according to the information of this user alone, the likelihood of each of these attribute values belonging to the targeted user decreases. Moreover, an attribute value should receive more votes if more users with higher vote capacities have the attribute value. Therefore, in Phase II, each social node votes for its attribute values via dividing its vote capacity among them, and each attribute value sums the vote capacities that are divided to it by its social neighbors. We treat the summed vote capacity of an attribute value as its similarity with v . Finally, we predict v has the attribute values that receive the highest votes.

4.2 Phase I

In Phase I, VIAL iteratively distributes a fixed vote capacity from the targeted user v to the rest of users. We denote by $\vec{s}_v^{(i)}$ the vote capacity vector in the i th iteration, where $\vec{s}_{vu}^{(i)}$ is the vote capacity of node u in the i th iteration. Initially, v has a vote capacity $|V_s|$ and all other social nodes have vote capacities of 0. Formally, we have:

$$\vec{s}_{vu}^{(0)} = \begin{cases} |V_s| & \text{if } u = v \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

In each iteration, VIAL applies three local rules. They are *dividing*, *backtracking*, and *aggregating*. Intuitively,

if a user u has more (hop-2) social neighbors, then each neighbor could receive less vote capacity from u . Therefore, our dividing rule splits a social node's vote capacity to its social neighbors and hop-2 social neighbors. The backtracking rule takes a portion of every social node's vote capacity and assigns them back to the targeted user v , which is based on the intuition that social nodes that are closer to v in the SBA network are likely to be more similar to v and should get more vote capacities. A user could have a higher vote capacity if it is linked to more social neighbors and hop-2 social neighbors with higher vote capacities. Thus, for each user u , the aggregating rule collects the vote capacities that are shared to u by its social neighbors and hop-2 social neighbors. Fig. 2 illustrates the three local rules. Next, we elaborate the three local rules.

Dividing: A social node u could have social neighbors, behavior-sharing social neighbors, and attribute-sharing social neighbors. To distinguish them, we use three weights w_S , w_{BS} , and w_{AS} to represent the shares of them, respectively. For instance, the total vote capacity shared to social neighbors of u in the t th iteration is $\vec{s}_{vu}^{(t-1)} \times \frac{w_S}{w_S + w_{BS} + w_{AS}}$. Then we further divide the vote capacity among each type of neighbors according to their link weights. We define $I_{u,Y} = 1$ if the set of neighbors $\Gamma_{u,Y}$ is non-empty, otherwise $I_{u,Y} = 0$, where $Y = S, BS, AS$. The variables $I_{u,S}$, $I_{u,BS}$, and $I_{u,AS}$ are used to consider the scenarios where u does not have some type(s) of neighbors, in which u 's vote capacity is divided among less than three types of social neighbors. For convenience, we denote $w_T = w_S I_{u,S} + w_{BS} I_{u,BS} + w_{AS} I_{u,AS}$.

- **Social neighbors.** A social neighbor $x \in \Gamma_{u,S}$ receives a higher vote capacity from u if their link weight (e.g., tie strength) is higher. Therefore, we model the vote capacity $p_v^{(i)}(u,x)$ that is divided to x by u in the i th iteration as:

$$p_v^{(i)}(u,x) = \vec{s}_{vu}^{(i-1)} \cdot \frac{w_S}{w_T} \cdot \frac{w_{ux}}{d_{u,S}}, \quad (2)$$

where $d_{u,S}$ is the summation of weights of social links that are incident from u .

- **Behavior-sharing social neighbors.** A behavior-sharing social neighbor $x \in \Gamma_{u,BS}$ receives a higher vote capacity from u if they share more behavior neighbors with higher predictiveness. Thus, we model vote capacity $q_v^{(i)}(u,x)$ that is divided to x by u in the i th iteration as:

$$q_v^{(i)}(u,x) = \vec{s}_{vu}^{(i-1)} \cdot \frac{w_{BS}}{w_T} \cdot w_B(u,x), \quad (3)$$

where $w_B(u,x) = \sum_{y \in \Gamma_{u,B} \cap \Gamma_{x,B}} \frac{w_{uy}}{d_{u,B}} \cdot \frac{w_{xy}}{d_{y,S}}$, representing the overall share of vote capacity that u divides

to x because of their common behavior neighbors. Specifically, $\frac{w_{uy}}{d_{u,B}}$ characterizes the fraction of vote capacity u divides to the behavior neighbor y and $\frac{w_{xy}}{d_{y,S}}$ characterizes the fraction of vote capacity y divides to x . Large weights of w_{uy} and w_{xy} indicate y is a predictive behavior of the attribute values of u and x , and having more such common behavior neighbors make x share more vote capacity from u .

- **Attribute-sharing social neighbors.** An attribute-sharing social neighbor $x \in \Gamma_{u,AS}$ receives a higher vote capacity from u if they share more attribute neighbors with higher degree of affinity. Thus, we model vote capacity $r_v^{(i)}(u,x)$ that is divided to x by u in the i th iteration as:

$$r_v^{(i)}(u,x) = \vec{s}_{vu}^{(i-1)} \cdot \frac{w_{AS}}{w_T} \cdot w_A(u,x), \quad (4)$$

where $w_A(u,x) = \sum_{y \in \Gamma_{u,A} \cap \Gamma_{x,A}} \frac{w_{uy}}{d_{u,A}} \cdot \frac{w_{xy}}{d_{y,S}}$, representing the overall share of vote capacity that u divides to x because of their common attribute neighbors. Specifically, $\frac{w_{uy}}{d_{u,A}}$ characterizes the fraction of vote capacity u divides to the attribute neighbor y and $\frac{w_{xy}}{d_{y,S}}$ characterizes the fraction of vote capacity y divides to x . Large weights of w_{uy} and w_{xy} indicate y is an attribute value with a high degree of affinity, and having more such common attribute values make x share more vote capacity from u .

We note that a social node x could be multiple types of social neighbors of u (e.g., x could be social neighbor and behavior-sharing social neighbor of u), in which x receives multiple shares of vote capacity from u and we sum them as x 's final share of vote capacity.

Backtracking: For each social node u , the backtracking rule takes a portion α of u 's vote capacity back to the targeted user v . Specifically, the vote capacity backtracked to v from u is $\alpha \vec{s}_{vu}^{(i-1)}$. Considering backtracking, the vote capacity divided to the social neighbor x of u in the dividing step is modified as $(1 - \alpha)p_v^{(i)}(u,x)$. Similarly, the vote capacities divided to a behavior-sharing social neighbor and an attribute-sharing social neighbor x are modified as $(1 - \alpha)q_v^{(i)}(u,x)$ and $(1 - \alpha)r_v^{(i)}(u,x)$, respectively. We call the parameter α *backtracking strength*. A larger backtracking strength enforces more vote capacity to be distributed among the social nodes that are closer to v in the SBA network. $\alpha = 0$ means no backtracking. We will show that, via both theoretical and empirical evaluations, VIAL achieves better accuracy with backtracking.

We can verify that $\sum_{x \in \Gamma_{u,S}} \frac{w_{ux}}{d_{u,S}} = 1$, $\sum_{x \in \Gamma_{u,BS}} w_B(u,x) = 1$, and $\sum_{x \in \Gamma_{u,AS}} w_A(u,x) = 1$ for every user u in the dividing step. In other words, every user divides all its

Algorithm 1: Phase I of VIAL

Input: $G = (V, E, w, t)$, \mathbf{M} , v , ε , and α .

Output: \vec{s}_v .

```

1 begin
2   //Initializing the vote capacity vector.
3   for  $u \in V_s$  do
4     if  $u = v$  then
5        $\vec{s}_{vu}^{(0)} \leftarrow |V_s|$ 
6     else
7        $\vec{s}_{vu}^{(0)} \leftarrow 0$ 
8     end
9   end
10   $error \leftarrow 1$ 
11  while  $error > \varepsilon$  do
12     $\vec{s}_v^{(i)} \leftarrow \alpha \vec{e}_v + (1 - \alpha) \mathbf{M}^T \vec{s}_v^{(i-1)}$ 
13     $error \leftarrow |\vec{s}_v^{(i)} - \vec{s}_v^{(i-1)}| / |V_s|$ 
14  end
15  return  $\vec{s}_v^{(i)}$ 
16 end

```

vote capacity to its neighbors (including the user itself if the user has hop-2 social neighbors). Therefore, the total vote capacity keeps unchanged in every iteration, and the vote capacity that is backtracked to the targeted user is $\alpha|V_s|$.

Aggregating: The aggregating rule computes a new vote capacity for u by aggregating the vote capacities that are divided to u by its neighbors in the i th iteration. For the targeted user v , we also collect the vote capacities that are backtracked from all social nodes. Formally, our aggregating rule is represented as Equation 5.

Matrix representation: We derive the Phase I of our attack using matrix terminologies, which makes it easier to iteratively compute the vote capacities. Towards this end, we define a *dividing matrix* $\mathbf{M} \in R^{|V_s| \times |V_s|}$, which is formally represented in Equation 6. The dividing matrix encodes the dividing rule. Specifically, u divides \mathbf{M}_{ux} fraction of its vote capacity to the neighbor x in the dividing step. Note that \mathbf{M} includes the dividing rule for all three types of social neighbors. With the dividing matrix \mathbf{M} , we can represent the backtracking and aggregating rules in the i th iteration as follows:

$$\vec{s}_v^{(i)} = \alpha \vec{e}_v + (1 - \alpha) \mathbf{M}^T \vec{s}_v^{(i-1)}, \quad (7)$$

where \vec{e}_v is a vector with the v th entry equals $|V_s|$ and all other entries equal 0, and \mathbf{M}^T is the transpose of \mathbf{M} .

Given an initial vote capacity vector specified in Equation 1, we iteratively apply Equation 7 until the difference between the vectors in two consecutive iterations is smaller than a predefined threshold. Algorithm 1 shows Phase I of our attack.

Our aggregating rule to compute the new vote capacity $\vec{s}_{vu}^{(i)}$ for u :

$$\vec{s}_{vu}^{(i)} = \begin{cases} (1 - \alpha)(\sum_{x \in \Gamma_{u,S}} p_v^{(i)}(x, u) + \sum_{x \in \Gamma_{u,BS}} q_v^{(i)}(x, u) + \sum_{x \in \Gamma_{u,AS}} r_v^{(i)}(x, u)) & \text{if } u \neq v \\ (1 - \alpha)(\sum_{x \in \Gamma_{u,S}} p_v^{(i)}(x, u) + \sum_{x \in \Gamma_{u,BS}} q_v^{(i)}(x, u) + \sum_{x \in \Gamma_{u,AS}} r_v^{(i)}(x, u)) + \alpha|V_s| & \text{otherwise} \end{cases} \quad (5)$$

Our dividing matrix:

$$\mathbf{M}_{ux} = \begin{cases} \delta_{ux,S} \cdot \frac{w_S}{w_T} \cdot \frac{w_{ux}}{d_{u,S}} + \delta_{ux,BS} \cdot \frac{w_{BS}}{w_T} \cdot w_B(u, x) + \delta_{ux,AS} \cdot \frac{w_{AS}}{w_T} \cdot w_A(u, x) & \text{if } x \in \Gamma_{u,S} \cup \Gamma_{u,BS} \cup \Gamma_{u,AS} \\ 0 & \text{otherwise,} \end{cases} \quad (6)$$

where $\delta_{ux,Y} = 1$ if $x \in \Gamma_{u,Y}$, otherwise $\delta_{ux,Y} = 0$, $Y = S, BS, AS$.

4.3 Phase II

In Phase I, we obtained a vote capacity for each user. On one hand, the targeted user could be more likely to share attribute values with the users with higher vote capacities. On the other hand, if a user has more attribute values, then the likelihood of each of these attribute values belonging to the targeted user could be smaller. For instance, if a user with a high vote capacity once studied in more universities for undergraduate education, then according to this user's information alone, the likelihood of the targeted user studying in each of those universities could be smaller.

Moreover, among a user's attribute values, an attribute value that has a higher degree of affinity (represented by the weight of the corresponding attribute link) with the user could be more likely to be an attribute value of the targeted user. For instance, suppose a user once lived in two cities, one of which is the user's hometown while the other of which is a city where the user once travelled; the user has a high vote capacity because he/she is structurally close (e.g., he/she shares many common friends with the targeted user) to the targeted user; then the targeted user is more likely to be from the hometown of the user than from the city the user once travelled.

Therefore, to capture these observations, we divide the vote capacity of a user to its attribute values in proportion to the weights of its attribute links; and each attribute value sums the vote capacities that are divided to it by the users having the attribute value. Intuitively, an attribute value receives more votes if more users with higher vote capacities link to the attribute value via links with higher weights. Formally, we have

$$\vec{t}_{va} = \sum_{u \in \Gamma_{a,S}} \vec{s}_{vu} \cdot \frac{w_{au}}{d_{u,A}}, \quad (8)$$

where \vec{t}_{va} is the final votes of the attribute value a , $\Gamma_{a,S}$ is the set of users who have the attribute value a , $d_{u,A}$ is the sum of weights of attribute links that are incident from u .

We treat the summed votes of an attribute value as its similarity with v . Finally, we predict v has the attribute values that receive the highest votes.

4.4 Confidence Estimation

For a targeted user, a *confidence estimator* takes the final votes for all attribute values as an input and produces a confidence score. A higher confidence score means that attribute inference for the targeted user is more trustworthy. We design a confidence estimator based on clustering techniques. A targeted user could have multiple attribute values for a single attribute, and our attack could produce close votes for these attribute values. Therefore, we design a confidence estimator called *clusterness* for our attack. Specifically, we first use a clustering algorithm (e.g., k-means [24]) to group the votes that our attack produces for all candidate attribute values into two clusters. Then we compute the average vote in each cluster, and the clusterness is the difference between the two average votes. The intuition of our clusterness is that if our attack successfully infers the targeted user's attribute values, there could be a cluster of attribute values whose votes are significantly higher than other attribute values'.

Suppose the attacker chooses a confidence threshold and only predicts attributes for targeted users whose confidence scores are higher than the threshold. Via setting a larger confidence threshold, the attacker will attack less targeted users but could achieve a higher success rate. In other words, an attacker can balance between the success rates and the number of targeted users to attack via confidence estimation.

5 Theoretical Analysis

We analyze the convergence of VIAL and derive the analytical forms of vote capacity vectors, discuss the importance of the backtracking rule, and analyze the complexity of VIAL.

5.1 Convergence and Analytical Solutions

We first show that for any backtracking strength $\alpha \in (0, 1]$, the vote capacity vectors converge.

Theorem 1. *For any backtracking strength $\alpha \in (0, 1]$, the vote capacity vectors $\vec{s}_v^{(0)}, \vec{s}_v^{(1)}, \vec{s}_v^{(2)}, \dots$ converge, and the converged vote capacity vector is $\alpha(I - (1 - \alpha)\mathbf{M}^T)^{-1}\vec{e}_v$. Formally, we have:*

$$\vec{s}_v = \lim_{i \rightarrow \infty} \vec{s}_v^{(i)} = \alpha(I - (1 - \alpha)\mathbf{M}^T)^{-1}\vec{e}_v, \quad (9)$$

where I is an identity matrix and $(I - (1 - \alpha)\mathbf{M}^T)^{-1}$ is the inverse of $(I - (1 - \alpha)\mathbf{M}^T)$.

Proof. See Appendix A. \square

Next, we analyze the convergence of VIAL and the analytical form of the vote capacity vector when the backtracking strength $\alpha = 0$.

Theorem 2. *When $\alpha = 0$ and the SBA network is connected, the vote capacity vectors $\vec{s}_v^{(0)}, \vec{s}_v^{(1)}, \vec{s}_v^{(2)}, \dots$ converge, and the converged vote capacity vector is proportional to the unique stationary distribution of the Markov chain whose transition matrix is \mathbf{M} . Mathematically, the converged vote capacity vector \vec{s}_v can be represented as:*

$$\vec{s}_v = |V_s| \vec{\pi}, \quad (10)$$

where $\vec{\pi}$ is the unique stationary distribution of the Markov chain whose transition matrix is \mathbf{M} .

Proof. See Appendix B. \square

With Theorem 2, we have the following corollary, which states that the vote capacity of a user is proportional to its weighted degree for certain assignments of the shares of social neighbors and hop-2 social neighbors in the dividing step.

Corollary 1. *When $\alpha = 0$, the SBA network is connected, and for each user u , the shares of social neighbors, behavior-sharing social neighbors, and attribute-sharing social neighbors in the dividing step are $w_S = \tau \cdot d_{u,S}$, $w_{BS} = \tau \cdot d_{u,B}$, and $w_{AS} = \tau \cdot d_{u,A}$, respectively, then we have:*

$$\vec{s}_{vu} = |V_s| \frac{d_u}{D}, \quad (11)$$

where τ is any positive number, d_u is the weights of all links of u and D is the twice of the total weights of all links in the SBA network, i.e., $D = \sum_u d_u$.

Proof. See Appendix C. \square

5.2 Importance of Backtracking

Theorem 2 implies that when there is no backtracking, the converged vote capacity vector is independent with the targeted users. In other words, VIAL with no backtracking predicts the same attribute values for all targeted users. This explains why VIAL with no backtracking achieves suboptimal performance. We will further empirically evaluate the impact of backtracking strength in our experiments, and we found that VIAL’s performance significantly degrades when there is no backtracking.

5.3 Time Complexity

The major cost of VIAL is from Phase I, which includes computing \mathbf{M} and iteratively computing the vote capacity vector. \mathbf{M} only needs to be computed once and is applied to all targeted users. \mathbf{M} is a sparse matrix with $O(m)$ non-zero entries, where m is the number of links in the SBA network. To compute \mathbf{M} , for every social node, we need to go through its social neighbors and hop-2 social neighbors; and for a hop-2 social neighbor, we need to go through the common attribute/behavior neighbors between the social node and the hop-2 social neighbor. Therefore, the time complexity of computing \mathbf{M} is $O(m)$.

Using sparse matrix representation of \mathbf{M} , the time complexity of each iteration (i.e., applying Equation 7) in computing the vote capacity vector is $O(m)$. Therefore, the time complexity of computing the vote capacity vector for one targeted user is $O(d \cdot m)$, where d is the number of iterations. Thus, the overall time complexity of VIAL is $O(d \cdot m)$ for one targeted user.

6 Data Collection

We collected a dataset from Google+ and Google Play to evaluate our VIAL attack and previous attacks. Specifically, we collected social structures and user attributes from Google+, and user review behaviors from Google Play. Google assigns each user a 21-digit universal ID, which is used in both Google+ and Google Play. We first collected a social network with user attributes from Google+ via iteratively crawling users’ friends. Then we crawled review data of users in the Google+ dataset. All the information that we collected is publicly available.

6.1 Google+ Dataset

Each user in Google+ has an outgoing friend list (i.e., “in your circles”), an incoming friend list (i.e., “have you in circles”), and a profile. Shortly after Google+ was launched in late June 2011, Gong et al. [16, 15] began to crawl daily snapshots of public Google+ social network structure and user profiles (e.g., *major*, *employer*,

and *cities lived*). Their dataset includes 79 snapshots of Google+ collected from July 6 to October 11, 2011. Each snapshot was a large Weakly Connected Component of Google+ social network at the time of crawling.

We obtained one collected snapshot from Gong et al. [16, 15]. To better approximate friendships between users, we construct an undirected social network from the crawled Google+ dataset via keeping an undirected link between a user u and v if u is in v 's both incoming friend list and outgoing friend list. After preprocessing, our Google+ dataset consists of 1,111,905 users and 5,328,308 undirected social links.

User attributes: We consider three attributes, *major*, *employer*, and *cities lived*. We note that, although we focus on these attributes that are available to us at a large scale, our attack is also applicable to infer other attributes such as sexual orientation, political views, and religious views. Moreover, some targeted users might not view inferring these attributes as an privacy attack, but an attacker can leverage these attributes to further link users across online social networks [4, 14, 2, 13] or even link them with offline records to perform more serious security and privacy attacks [38, 32].

We take the strings input by a user in its Google+ profile as attribute values. We found that most attribute values are owned by a small number of users while some are owned by a large number of users. Users could fill in their profiles freely in Google+, which could be one reason that we observe many infrequent attribute values. Specifically, different users might have different names for the same attribute value. For instance, the major of Computer Science could also be abbreviated as CS by some users. Indeed, we find that 20,861 users have Computer Science as their major and 556 users have CS as their major in our dataset. Moreover, small typos (e.g., one letter is incorrect) in the free-form inputs make the same attribute value be treated as different ones. Therefore, we manually label a set of attribute values.

1) *Major*. We consider the top-100 majors that are claimed by the most users. We manually merge the majors that actually refer to the same one, e.g., Computer Science and CS, BTech and Biotechnology. After preprocessing, we obtain 62 distinct majors. 8.4% of users in our dataset have at least one of these majors.

2) *Employer*. Similar to major, we select the top-100 employers that are claimed by the most users and manually merge the employers that refer to the same one. We obtain 78 distinct employers, and 3.1% of users have at least one of these employers.

3) *Cities lived*. Again, we select the top-100 cities in which most users in the Google+ dataset claimed they have lived in. After we manually merge the cities that actually refer to the same one, we obtain 70 distinct cities. 8% of users have at least one of these attribute values.

Table 1: Basic statistics of our SBA.

#nodes			#links		
social	behavior	attri.	social	behavior	attri.
1,111,905	48,706	210	5,328,308	3,635,231	269,997

Summary and limitations: In total, we consider 210 popular distinct attribute values, including 62 majors, 78 employers, and 70 cities. We acknowledge that our Google+ dataset might not be a representative sample of the recent entire Google+ social network, and thus the inference attack success rates obtained in our experiments might not represent those of the entire Google+ social network.

6.2 Crawling Google Play

There are 7 categories of items in Google Play. They are *apps*, *tv*, *movies*, *music*, *books*, *newsstand*, and *devices*. Google Play provides two review mechanisms for users to provide feedback on an item. They are the *liking* mechanism and the *rating* mechanism. In the liking mechanism, a user simply clicks a like button to express his preference about an item. In the rating mechanism, a user gives a rating score which is in the set {1,2,3,4,5} as well as a detailed comment to support his/her rating. A score of 1 represents low preference and a score of 5 represents high preference. We call a user *reviewed* an item if the user rated or liked the item.

User reviews are publicly available in Google Play. Specifically, after a user u logs in Google Play, u can view the list of items reviewed by any user v once u can obtain v 's Google ID. We crawled the list of items re-reviewed by each user in the Google+ dataset.

We find that 33% of users in the Google+ dataset have reviewed at least one item. In total, we collected 260,245 items and 3,954,822 reviews. Since items with too few reviews might not be informative to distinguish users with different attribute values, we use items that were re-reviewed by at least 5 users. After preprocessing, we have 48,706 items and 3,635,231 reviews.

6.3 Constructing SBA Networks

We take each user in the Google+ dataset as a social node and links between them as social links. For each item in our Google Play dataset, we add a corresponding behavior node. If a user reviewed an item, we create a link between the corresponding social node and the corresponding behavior node. That a user reviewed an item means that the user once used the item. Using similar items could indicate similar interests, user characteristics, and user attributes. To predict attribute values, we further

add additional attribute nodes to represent attribute values, and we create a link between a social node and an attribute node if the user has the attribute value. Table 1 shows the basic statistics of our constructed SBA for predicting attribute values.

In this work, we set the weights of all links in the SBA to be 1. Therefore, our attacking result represents a lower bound on what an attacker can achieve in practice. An attacker could leverage machine learning techniques (we discuss one in Section 8) to learn link weights to further improve success rates.

7 Experiments

7.1 Experimental Setup

We describe the metrics we adopt to evaluate various attacks, training and testing, and parameter settings.

Evaluation metrics: All attacks we evaluate essentially assign a score for each candidate attribute value. Given a targeted user v , we predict top- K candidate attribute values that have the highest scores for each attribute including major, employer, and cities lived. We use Precision, Recall, and F-score to evaluate the top- K predictions. In particular, Precision is the fraction of predicted attribute values that belong to v . Recall is the fraction of v 's attribute values that are among the predicted K attribute values. We address score ties in the manner described by McSherry and Najork [31]. Precision characterizes how accurate an attacker's inferences are while Recall characterizes how many user attributes are correctly inferred by an attacker. In particular, Precision for top-1 prediction is the fraction of users that the attacker can correctly infer at least one attribute value. F-score is the harmonic mean of Precision and Recall, i.e., we have

$$\text{F-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Moreover, we average the three metrics over all targeted users. For convenience, we will also use P, R, and F to represent Precision, Recall, and F-Score, respectively.

We also define *performance gain* and *relative performance gain* of one attack \mathcal{A} over another attack \mathcal{B} to compare their relative performances. We take Precision as an example to show their definitions as follows:

Performance gain:

$$\Delta P = \text{Precision}_{\mathcal{A}} - \text{Precision}_{\mathcal{B}}$$

Relative performance gain:

$$\Delta P\% = \frac{\text{Precision}_{\mathcal{A}} - \text{Precision}_{\mathcal{B}}}{\text{Precision}_{\mathcal{B}}} \times 100\%$$

Training and testing: For each attribute value, we sample 5 users uniformly at random from the users that have the attribute value and have reviewed at least 5 items, and we treat them as test (i.e., targeted) users. In total, we have around 1,050 test users. For test users, we remove their attribute links from the SBA network and use them as groundtruth. We repeat the experiments 10 times and average the evaluation metrics over the 10 trials.

Parameter settings: In the dividing step, we set equal shares for social neighbors, behavior-sharing social neighbors, and attribute-sharing social neighbors, i.e., $w_S = w_{BS} = w_{AS} = \frac{1}{3}$. The number of iterations to compute the vote capacity vector is $d = \lfloor \log |V_s| \rfloor = 20$, after which the vote capacity vector converges. Unless otherwise stated, we set the backtracking strength $\alpha = 0.1$.

7.2 Compared Attacks

We compare VIAL with friend-based attacks, behavior-based attacks, and attacks that use both social structures and behaviors. These attacks essentially assign a score for each candidate attribute value, and return the K attribute values that have the highest scores. Suppose v is a test user and a is an attribute value, and we denote by $S(v, a)$ the score assigned to a for v .

Random: This baseline method computes the fraction of users in the training dataset that have a certain attribute value a , and it treats such fraction as the score $S(v, a)$ for all test users.

Friend-based attacks: We compare with three friend-based attacks, i.e., CN-SAN, AA-SAN, and RWwR-SAN [15]. They were shown to outperform previous attacks such as LINK [46, 15].

- **CN-SAN.** $S(v, a)$ is the number of common social neighbors between v and a .
- **AA-SAN.** This attack weights the importance of each common social neighbor between v and a proportional to the inverse of the log of its number of neighbors. Formally, $S(v, a) = \sum_{u \in \Gamma_{v,S} \cap \Gamma_{a,S}} \frac{1}{\log |\Gamma_u|}$.
- **RWwR-SAN.** RWwR-SAN augments the social network with additional attribute nodes. Then it performs a random walk that is initialized from the test user v on the augmented graph. The stationary probability of the attribute node that corresponds to a is treated as the score $S(v, a)$.

Behavior-based attacks: We also evaluate three behavior-based attacks.

- **Logistic regression (LG-B-I) [42].** LG-B-I treats each attribute value as a class and learns a multi-class logistic regression classifier with the training

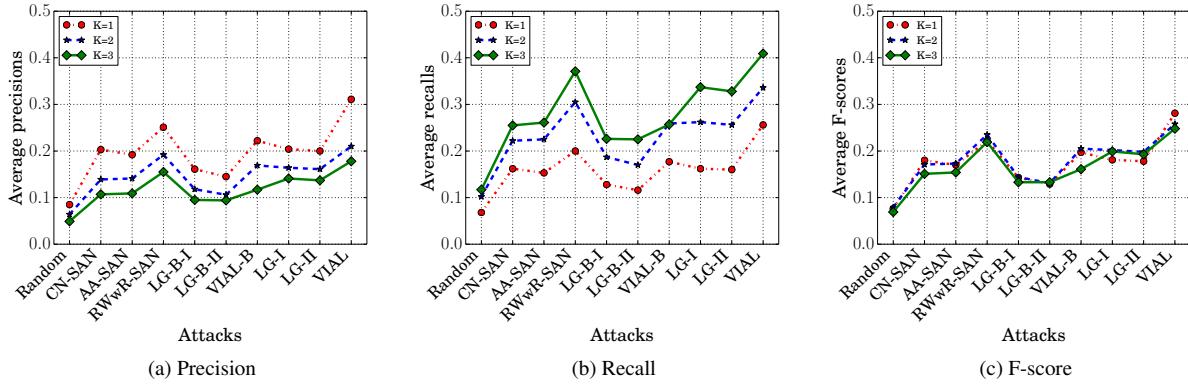


Figure 3: Precision, Recall, and F-Score for inferring majors. Although these attacks do not have temporal orderings, we connect them via curves in the figures to better contrast them.

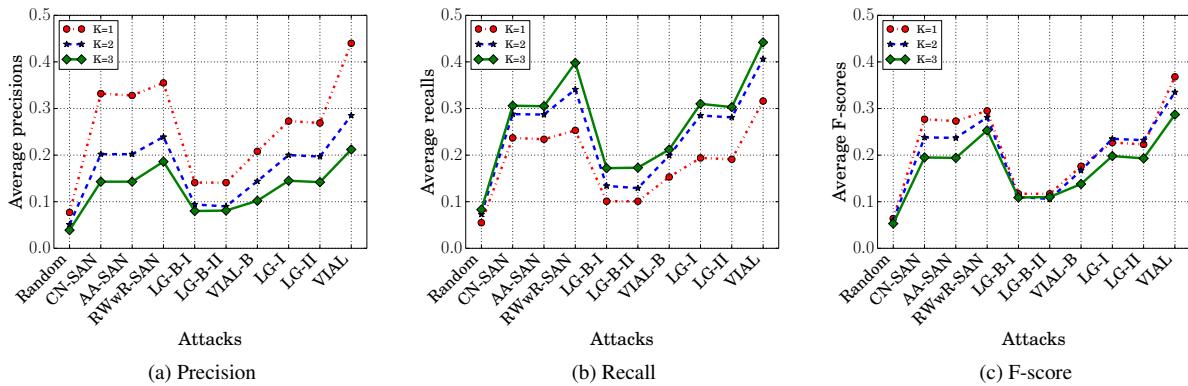


Figure 4: Precision, Recall, and F-Score for inferring employers.

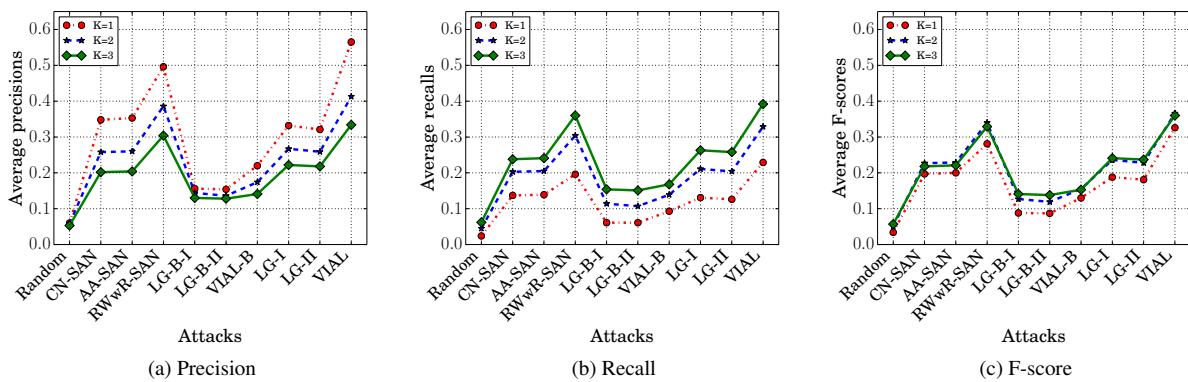


Figure 5: Precision, Recall, and F-Score for inferring cities.

Table 2: Performance gains and relative performance gains of RWwR-SAN over other friend-based attacks, where $K = 1$. Results are averaged over all attributes. We find that RWwR-SAN is the best friend-based attack.

Attack	ΔP	$\Delta P\%$	ΔR	$\Delta R\%$	ΔF	$\Delta F\%$
CN-SAN	0.07	24%	0.04	24%	0.05	24%
AA-SAN	0.08	26%	0.04	26%	0.05	26%

Table 3: Performance gains and relative performance gains of VIAL-B over other behavior-based attacks, where $K = 1$. We find that VIAL-B is the best behavior-based attack.

Attack	ΔP	$\Delta P\%$	ΔR	$\Delta R\%$	ΔF	$\Delta F\%$
LG-B-I	0.06	42%	0.04	47%	0.05	45%
LG-B-II	0.07	47%	0.05	52%	0.06	50%

dataset. Specifically, LG-B-I extracts a feature vector whose length is the number of items for each user that has review data, and a feature has a value of the rating score that the user gave to the corresponding item. Google Play allows users to rate or like an item, and we treat a liking as a rating score of 5. For a test user, the learned logistic regression classifier returns a posterior probability distribution over the possible attribute values, which are used as the scores $S(v, a)$. Weinsberg et al. [42] showed that logistic regression classifier outperforms other classifiers including SVM [10] and Naive Bayes [29].

- **Logistic regression with binary features (LG-B-II) [25].** The difference between LG-B-II and LG-B-I is that LG-B-II extracts binary feature vectors for users. Specifically, a feature has a value of 1 if the user has reviewed the corresponding item.
- **VIAL-B.** A variant of VIAL that only uses behavior data. Specifically, we remove social links from the SBA network and perform our VIAL attack using the remaining links.

Attacks combining social structures and behaviors: Intuitively, we can combine social structures and behaviors via concatenating social structure features with behavior features. We compare with two such attacks.

- **Logistic regression (LG-I).** LG-I extracts a binary feature vector whose length is the number of users from social structures for each user, and a feature has a value of 1 if the user is a friend of the person that corresponds to the feature. Then LG-I concatenates this feature vector with the one used in LG-B-I and learns multi-class logistic regression classifiers.

Table 4: Performance gains and relative performance gains of VIAL over other attacks combining social structures and behaviors, where $K = 1$. We find that VIAL substantially outperforms other attacks.

Attack	ΔP	$\Delta P\%$	ΔR	$\Delta R\%$	ΔF	$\Delta F\%$
LG-I	0.17	61%	0.10	65%	0.13	63%
LG-II	0.18	65%	0.11	69%	0.13	67%

Table 5: Performance gains and relative performance gains of VIAL over Random, RWwR-SAN (the best friend-based attack), and VIAL-B (the best behavior-based attack), where $K = 1$.

Attack	ΔP	$\Delta P\%$	ΔR	$\Delta R\%$	ΔF	$\Delta F\%$
Random	0.36	526%	0.22	535%	0.27	534%
RWwR-SAN	0.07	20%	0.05	23%	0.06	22%
VIAL-B	0.22	102%	0.13	99%	0.16	100%

- **Logistic regression with binary features (LG-II).** LG-II concatenates the binary social structure feature vector with the binary behavior feature vector used by LG-B-II.

We use the popular package LIBLINEAR [12] to learn logistic regression classifiers.

7.3 Results

Fig. 3-Fig. 5 demonstrate the Precision, Recall, and F-score for top- K inference of major, employer, and city, where $K = 1, 2, 3$. Table 2-Table 5 compare different attacks using results that are averaged over all attributes. Our metrics are averaged over 10 trials. We find that standard deviations of the metrics are very small, and thus we do not show them for simplicity. Next, we describe several key observations we have made from these results.

Comparing friend-based attacks: We find that RWwR-SAN performs the best among the friend-based attacks. Our observation is consistent with the previous work [15]. To better illustrate the difference between the friend-based attacks, we show the performance gains and relative performance gains of RWwR-SAN over other friend-based attacks in Table 2. Please refer to Section 7.1 for formal definitions of (relative) performance gains. The (relative) performance gains are averaged over all attributes (i.e., major, employer, and city). The reason why RWwR-SAN outperforms other friend-based attacks is that RWwR-SAN performs a random walk among the augmented graph, which better leverages the graph structure, while other attacks simply count the number of common neighbors or weighted common neighbors.

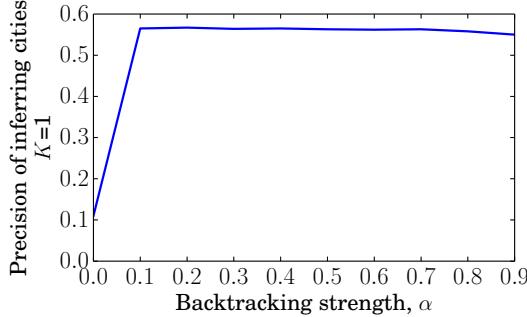


Figure 6: Impact of the backtracking strength on the Precision of VIAL for inferring cities. We observe that backtracking substantially improves VIAL’s performance.

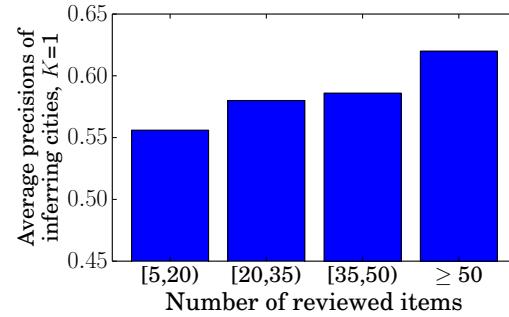


Figure 7: Impact of the number of reviewed items on the Precision of our attack VIAL for inferring cities. We observe that, when users share more behaviors, our attack is able to more accurately predict their attributes.

Comparing behavior-based attacks: We find that VIAL-B performs the best among the behavior-based attacks. Table 3 shows the average performance gains and relative performance gains of VIAL-B over other behavior-based attacks. Our results indicate that our graph-based attack is a better way to leverage behavior structures, compared to LG-B-I and LG-B-II, which flatten the behavior structures into feature vectors. Moreover, LG-B-I and LG-B-II achieve very close performances, which indicates that the rating scores carry little information about user attributes.

Comparing attacks combining social structure and behavior: We find that VIAL performs the best among the attacks combining social structures and behaviors. Table 4 shows the average performance gains and relative performance gains of VIAL over other attacks. Our results imply that, compared to flattening the structures into feature vectors, our graph-based attack can better integrate social structures and user behaviors.

Comparing VIAL with the best friend-based attack and the best behavior-based attack: Table 4 shows the average performance gains and relative performance gains of VIAL over Random, the best friend-based attack, and the best behavior-based attack. We find that VIAL significantly outperforms these attacks, indicating the importance of combining social structures and behaviors to perform attribute inference. This implies that, when an attacker wants to attack user privacy via inferring their private attributes, the attacker can successfully attack substantially more users using VIAL.

Impact of backtracking strength: Fig. 6 shows the impact of backtracking strength on the Precision of VIAL for inferring cities. According to Theorem 1, VIAL with $\alpha = 1$ reduces to random guessing, and thus we do not show the corresponding result in the figure. $\alpha = 0$ corresponds to the case in which VIAL does not use backtracking. We observe that not using backtracking sub-

stantially decreases the performance of VIAL. The reason might be that 1) $\alpha = 0$ makes VIAL predict the same attribute values for all test users, according to Theorem 2, and 2) a user’ attributes are close to the user in the SBA network and backtracking makes it more likely for votes to be distributed among these attribute nodes. Moreover, we find that inference accuracies are stable across different backtracking strengths once they are larger than 0. The reason is that when we increase the backtracking strength, attribute values receive different votes, but the ones with top ranked votes only change slightly. We observe similar results for other attributes.

Impact of the number of reviewed items: Figure 7 shows the Precision as a function of the number of reviewed items for inferring cities lived. We average Precisions for test users whose number of reviewed items falls under a certain interval (i.e., [5,20), [20,35), [35,50), or ≥ 50). We observe that our attack can more accurately infer attributes for users who share more digital behaviors (i.e., reviewed items in our case).

Confidence estimation: Figure 8 shows the trade-off between the Precision and the fraction of users that are attacked via our confidence estimator. We observe that an attacker can increase the Precision ($K = 1$) of inferring cities from 0.57 to over 0.92 if the attacker attacks a half of the test users that are selected via confidence estimation. We also tried the confidence estimator called *gap statistic* [34], in which the confidence score for a targeted user is the difference between the score of the highest ranked attribute value and the score of the second highest ranked one. Our confidence estimator slightly outperforms gap statistic because a test user could have multiple attribute values and our attack could produce close scores for them.

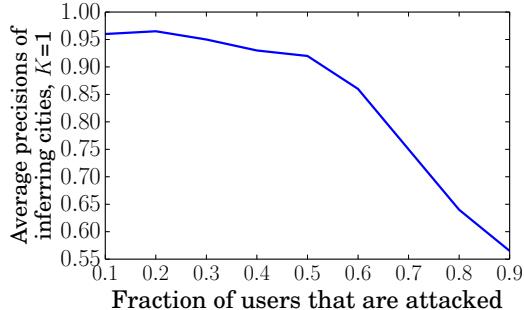


Figure 8: Confidence estimation: trade-off between the Precision of our attack and the fraction of test users that are attacked. An attacker can substantially improve the attack success rates when attacking less users that are selected by our confidence estimator.

8 Discussion

This work focuses on propagating vote capacity among the SBA network with given link weights, and our method VIAL is applicable to any link weights. However, it is an interesting future work to learn the link weights, which could further improve the attacker’s success rates. In the following, we discuss one possible approach to learn link weights. Phase I of VIAL essentially iteratively computes the vote capacity vector according to Equation 7. Therefore, Phase 1 of VIAL can be viewed as performing a *random walk with a restart* [40] on the subgraph consisting of social nodes and social links, where the matrix \mathbf{M}^T and α are the transition matrix and restart probability of the random walk, respectively. Therefore, the attacker could adapt *supervised random walk* [3] to learn the link weights. Specifically, the attacker already has a set of users with publicly available attributes and the attacker can use them as a training dataset to learn the link weights; the attacker removes these attributes from the SBA network as ground truth, and the link weights are learnt such that VIAL can predict attributes for these users the most accurately.

9 Related Work

Friend-based attribute inference: He et al. [18] transformed attribute inference to Bayesian inference on a Bayesian network that is constructed using the social links between users. They evaluated their method using a LiveJournal social network dataset with *synthesized* user attributes. Moreover, it is well known in the machine learning community that Bayesian inference is not scalable. Lindamood et al. [27] modified Naive Bayes classifier to incorporate social links and other attributes of users to infer some attribute. For instance, to infer

a user’s major, their method used the user’s other attributes such as employer and cities lived, the user’s social friends and their attributes. However, their approach is not applicable to users that share no attributes at all.

Zheleva and Getoor [46] studied various approaches to consider both social links and groups that users joined to perform attribute inference. They found that, with only social links, the approach LINK achieves the best performance. LINK represents each user as a binary feature vector, and a feature has a value of 1 if the user is a friend of the person that corresponds to the feature. Then LINK learns classifiers for attribute inference using these feature vectors. Gong et al. [15] transformed attribute inference to a link prediction problem. Moreover, they showed that their approaches CN-SAN, AA-SAN, and RWWR-SAN outperform LINK.

Mislove et al. [33] proposed to identify a local community in the social network by taking some seed users that share the same attribute value, and then they predicted all users in the local community to have the shared attribute value. Their approach is not able to infer attributes for users that are not in any local communities. Moreover, this approach is data dependent since detected communities might not correlate with the attribute value. For instance, Trauda et al. [41] found that communities in a MIT male network are correlated with residence but a female network does not have such property.

Thomas et al. [39] studied the inference of attributes such as gender, political views, and religious views. They used multi-label classification methods and leveraged features from users’ friends and wall posts. Moreover, they proposed the concept of multi-party privacy to defend against attribute inference.

Behavior-based attribute inference: Weinsberg et al. [42] investigated the inference of gender using the rating scores that users gave to different movies. In particular, they constructed a feature vector for each user; the i th entry of the feature vector is the rating score that the user gave to the i th movie if the user reviewed the i th movie, otherwise the i th entry is 0. They compared a few classifiers including Logistic Regression (LG) [22], SVM [10], and Naive Bayes [29], and they found that LG outperforms the other approaches. Bhagat et al. [6] studied attribute inference in an active learning framework. Specifically, they investigated which movies we should ask users to review in order to improve the inference accuracy the most. However, this approach might not be applicable in real-world scenarios because users might not be interested in reviewing the selected movies.

Chaabane et al. [8] used the information about the musics users like to infer attributes. They augmented the musics with the corresponding Wikipedia pages and then used topic modeling techniques to identify the latent similarities between musics. A user is predicted to share

attributes with those that like similar musics with the user. Kosinski et al. [25] tried to infer various attributes based on the list of pages that users liked on Facebook. Similar to the work performed by Weinsberg et al. [42], they constructed a feature vector from the Facebook likes and used Logistic Regression to train classifiers to distinguish users with different attributes. Luo et al. [28] constructed a model to infer household structures using users' viewing behaviors in Internet Protocol Television (IPTV) systems, and they showed promising results.

Other approaches: Bonneau et al. [7] studied the extraction of private user data in online social networks via various attacks such as account compromise, malicious applications, and fake accounts. These attacks can not infer user attributes that users do not provide in their profiles, while our attack can. Otterbacher [35] studied the inference of gender using users' writing styles. Zamal et al. [45] used a user's tweets and her neighbors' tweets to infer attributes. They didn't consider social structures nor user behaviors. Gupta et al. [17] tried to infer interests of a Facebook user via sentiment-oriented mining on the Facebook pages that were liked by the user. Zhong et al. [47] demonstrated the possibility of inferring user attributes using the list of locations where the user has checked in. These studies are orthogonal to ours since they exploited information sources other than the social structures and behaviors that we focus on.

Attribute inference using social structure and behavior could also be solved by a social recommender system (e.g., [44]). However, such approaches have higher computational complexity than our method for attacking a targeted user, and it is challenging for them to have theoretical guarantees as our attack. For instance, the approach proposed by Ye et al. [44] has a time complexity of $O(m \cdot k \cdot f \cdot d)$ on a single machine, where m is the number of edges, k is the latent topic size, f is the average number of friends, and d is the number of iterations. Note that both our VIAL and this approach can be parallelized on a cluster.

10 Conclusion and Future Work

In this work, we study the problem of attribute inference via combining social structures and user behaviors that are publicly available in online social networks. To this end, we first propose a *social-behavior-attribute (SBA)* network model to gracefully integrate social structures, user behaviors, and their interactions with user attributes. Based on the SBA network model, we design a *vote distribution attack (VIAL)* to perform attribute inference. We demonstrate the effectiveness of our attack both theoretically and empirically. In particular, via empirical evaluations on a real-world large scale dataset with 1.1

million users, we find that attribute inference is a serious practical privacy attack to online social network users and an attacker can successfully attack more users when considering both social structures and user behaviors. The fundamental reason why our attack succeeds is that private user attributes are statistically correlated with publicly available information, and our attack captures such correlations to map publicly available information to private user attributes.

A few interesting directions for future work include learning the link weights of a SBA network, generalizing VIAL to infer hidden social relationships between users, as well as defending against our inference attacks.

11 Acknowledgements

We would like to thank the anonymous reviewers for their insightful feedback. This work is supported by College of Engineering, Department of Electrical and Computer Engineering of the Iowa State University.

References

- [1] "Data brokers: a call for transparency and accountability," *Federal Trade Commission*, 2014.
- [2] S. Afroz, A. Caliskan-Islam, A. Stolerman, R. Greenstadt, and D. McCoy, "Doppelgänger finder: Taking stylometry to the underground," in *IEEE S & P*, 2014.
- [3] L. Backstrom and J. Leskovec, "Supervised random walks: predicting and recommending links in social networks," in *WSDM*, 2011.
- [4] S. Bartunov, A. Korshunov, S.-T. Park, W. Ryu, and H. Lee, "Joint link-attribute user identity resolution in online social networks," in *SNA-KDD*, 2012.
- [5] E. Behrends, *Introduction to Markov chains*. Vieweg, 2000.
- [6] S. Bhagat, U. Weinsberg, S. Ioannidis, and N. Taft, "Recommending with an agenda: Active learning of private attributes using matrix factorization," in *RecSys*, 2014.
- [7] J. Bonneau, J. Anderson, and G. Danezis, "Prying data out of a social network," in *ASONAM*, 2009.
- [8] A. Chaabane, G. Acs, and M. A. Kaafar, "You are what you like! information leakage through users' interests," in *NDSS*, 2012.
- [9] T. Chen, R. Boreli, M. A. Kâafar, and A. Friedman, "On the effectiveness of obfuscation techniques in online social networks," in *PETS*, 2014.

- [10] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, 1995.
- [11] R. Dey, C. Tang, K. Ross, and N. Saxena, "Estimating age privacy leakage in online social networks," in *INFOCOM*, 2012.
- [12] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *The Journal of Machine Learning Research*, 2008.
- [13] O. Goga, H. Lei, S. H. K. Parthasarathi, G. Friedland, R. Sommer, and R. Teixeira, "Exploiting innocuous activity for correlating users across sites," in *WWW*, 2013.
- [14] O. Goga, D. Perito, H. Lei, R. Teixeira, and R. Sommer, "Large-scale correlation of accounts across social networks," ICSI, Tech. Rep., 2013.
- [15] N. Z. Gong, A. Talwalkar, L. Mackey, L. Huang, E. C. R. Shin, E. Stefanov, E. R. Shi, and D. Song, "Joint link prediction and attribute inference using a social-attribute network," *ACM TIST*, 2014.
- [16] N. Z. Gong, W. Xu, L. Huang, P. Mittal, E. Stefanov, V. Sekar, and D. Song, "Evolution of social-attribute networks: Measurements, modeling, and implications using google+," in *IMC*, 2012.
- [17] P. Gupta, S. Gottipati, J. Jiang, and D. Gao, "Your love is public now: Questioning the use of personal information in authentication," in *AsiaCCS*, 2013.
- [18] J. He, W. W. Chu, and Z. V. Liu, "Inferring privacy information from social networks," in *IEEE Intelligence and Security Informatics*, 2006.
- [19] R. Heatherly, M. Kantacioglu, and B. Thuraisingham, "Preventing private information inference attacks on social networks," *IEEE TKDE*, 2013.
- [20] M. Humbert, T. Studer, M. Grossglauser, and J.-P. Hubaux, "Nowhere to hide: Navigating around privacy in online social networks," in *ESORICS*, 2013.
- [21] C. Jernigan and B. F. Mistree, "Gaydar: Facebook friendships expose sexual orientation," *First Monday*, vol. 14, no. 10, 2009.
- [22] D. W. H. Jr and S. Lemeshow, *Applied logistic regression*. John Wiley & Sons, 2004.
- [23] D. Jurgens, T. Finnethy, J. McCorriston, Y. T. Xu, and D. Ruths, "Geolocation prediction in twitter using social networks: A critical analysis and review of current practice," in *ICWSM*, 2015.
- [24] T. Kanungo, D. Mount, N. Netanyahu, and C. Piatko, "An efficient k-means clustering algorithm: analysis and implementation," *IEEE TPAMI*, 2002.
- [25] M. Kosinski, D. Stillwell, and T. Graepel, "Private traits and attributes are predictable from digital records of human behavior," *PNAS*, 2013.
- [26] S. Labitzke, F. Werling, and J. Mittag, "Do online social network friends still threaten my privacy?" in *CODASPY*, 2013.
- [27] J. Lindamood, R. Heatherly, M. Kantacioglu, and B. Thuraisingham, "Inferring private information using social network data," in *WWW*, 2009.
- [28] D. Luo, H. Xu, H. Zha, J. Du, R. Xie, X. Yang, and W. Zhang, "You are what you watch and when you watch: Inferring household structures from iptv viewing data," *IEEE Transactions on Broadcasting*, 2014.
- [29] A. McCallum and K. Nigam, "A comparison of event models for naive bayes text classification," in *AAAI*, 1998.
- [30] M. McPherson, L. Smith-Lovin, and J. M. Cook, "Birds of a feather: Homophily in social networks," *Annual Review of Sociology*, 2001.
- [31] F. McSherry and M. Najork, "Computing information retrieval performance measures efficiently in the presence of tied scores," in *ECIR*, 2008.
- [32] T. Minkus, Y. Ding, R. Dey, and K. W. Ross, "The city privacy attack: Combining social media and public records for detailed profiles of adults and children," in *COSN*, 2015.
- [33] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel, "You are who you know: Inferring user profiles in online social networks," *WSDM*, 2010.
- [34] A. Narayanan, H. Paskov, N. Z. Gong, J. Bethencourt, R. Shin, E. Stefanov, and D. Song, "On the feasibility of internet-scale author identification," in *IEEE S & P*, 2012.
- [35] J. Otterbacher, "Inferring gender of movie reviewers: exploiting writing style, content and metadata," in *CIKM*, 2010.
- [36] O. Perron, "Zur theorie der matrizes," *Mathematische Annalen*, 1907.
- [37] Spear Phishing Attacks, "<http://www.microsoft.com/protect/yourself/phishing/spear.mspx>".

- [38] L. Sweeney, “k-anonymity: a model for protecting privacy,” *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 2002.
- [39] K. Thomas, C. Grier, and D. M. Nicol, “unfriendly: Multi-party privacy risks in social networks,” in *PETS*, 2010.
- [40] H. Tong, C. Faloutsos, and J.-Y. Pan, “Fast random walk with restart and its applications,” in *ICDM*, 2006.
- [41] A. L. Trauda, P. J. Muchaa, and M. A. Porter, “Social structure of facebook networks,” *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 16, 2012.
- [42] U. Weinsberg, S. Bhagat, S. Ioannidis, and N. Taft, “Blurme: Inferring and obfuscating user gender based on ratings,” in *RecSys*, 2012.
- [43] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, “Identifying diverse usage behaviors of smartphone apps,” in *IMC*, 2011.
- [44] M. Ye, X. Liu, and W.-C. Lee, “Exploring social influence for recommendation - a probabilistic generative model approach,” in *SIGIR*, 2012.
- [45] F. A. Zamal, W. Liu, and D. Ruths, “Homophily and latent attribute inference: Inferring latent attributes of twitter users from neighbors,” in *ICWSM*, 2012.
- [46] E. Zheleva and L. Getoor, “To join or not to join: The illusion of privacy in social networks with mixed public and private user profiles,” in *WWW*, 2009.
- [47] Y. Zhong, N. J. Yuan, W. Zhong, F. Zhang, and X. Xie, “You are where you go: Inferring demographic attributes from location check-ins,” in *WSDM*, 2015.

A Proof of Theorem 1

According to Equation 7, we have:

$$\vec{s}_v^{(i)} = (1 - \alpha)^i (\mathbf{M}^T)^i \vec{s}_v^{(0)} + \alpha \left(\sum_{k=0}^{i-1} (1 - \alpha)^k (\mathbf{M}^T)^k \right) \vec{e}_v. \quad (12)$$

Therefore,

$$\begin{aligned} \lim_{i \rightarrow \infty} \vec{s}_v^{(i)} &= \lim_{i \rightarrow \infty} \alpha \left(\sum_{k=0}^{i-1} (1 - \alpha)^k (\mathbf{M}^T)^k \right) \vec{e}_v \\ &= \alpha (I - (1 - \alpha) \mathbf{M}^T)^{-1} \vec{e}_v. \end{aligned} \quad (13)$$

We note that the matrix $(I - (1 - \alpha) \mathbf{M}^T)$ is nonsingular because it is strictly diagonally dominant.

B Proof of Theorem 2

The matrix \mathbf{M} has non-negative entries, and each row of \mathbf{M} sums to be 1. Therefore, \mathbf{M} can be viewed as a transition matrix. In particular, \mathbf{M} can be viewed as a transition matrix of the following Markov chain on the SBA network: each social node is a state of the Markov chain; the transition probability from a social node u to another social node x is \mathbf{M}_{ux} , i.e., a social node u can only transit to its social neighbors or hop-2 social neighbors with non-zero probabilities.

When the SBA network is connected, the above Markov chain is *irreducible* and *aperiodic*. Therefore, the Markov chain has a unique stationary distribution $\vec{\pi}$. Moreover, according to the Perron-Frobenius theorem [36], we have:

$$\lim_{i \rightarrow \infty} (\mathbf{M}^T)^i = [\vec{\pi} \ \vec{\pi} \cdots \vec{\pi}]$$

When $\alpha = 0$, we have $\vec{s}_v^{(i)} = (\mathbf{M}^T)^i \vec{s}_v^{(0)}$. Thus, we have

$$\begin{aligned} \vec{s}_v &= \lim_{i \rightarrow \infty} \vec{s}_v^{(i)} \\ &= \lim_{i \rightarrow \infty} (\mathbf{M}^T)^i \vec{s}_v^{(0)} \\ &= [\vec{\pi} \ \vec{\pi} \cdots \vec{\pi}] \vec{s}_v^{(0)} \\ &= |V_s| \vec{\pi}, \end{aligned}$$

where $|V_s|$ is the sum of the entries of $\vec{s}_v^{(0)}$.

C Proof of Corollary 1

When $w_S = \tau \cdot d_{u,S}$, $w_{BS} = \tau \cdot d_{u,B}$, and $w_{AS} = \tau \cdot d_{u,A}$ for each user u , the Markov chain defined by the transition matrix \mathbf{M} is a random walk on a weighted graph $G_w = (V_w, E_w)$, which is defined as follows: $V_w = V_s$, an edge (u, x) in E_w means that x is u 's social neighbor or hop-2 social neighbor in the SBA network, and the weight of the edge $(u, x) \in E_w$ is $\delta_{ux,S} \cdot w_{ux} + \delta_{ux,BS} \cdot d_{u,B} \cdot w_B(u, x) + \delta_{ux,AS} \cdot d_{u,A} \cdot w_A(u, x)$. We can verify that, on the graph G_w , the weights of all edges that are incident to a node u sum to d_u . Therefore, the stationary distribution $\vec{\pi}$ [5] of the random walk on G_w is:

$$\vec{\pi} = \left[\frac{d_{u_1}}{D} \ \frac{d_{u_2}}{D} \cdots \frac{d_{u_{|V_s|}}}{D} \right]^T. \quad (14)$$

Thus, according to Theorem 2, we have $\vec{s}_{vu} = |V_s| \frac{d_u}{D}$.

Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016

Adam Lerner*, Anna Kornfeld Simpson*, Tadayoshi Kohno, Franziska Roesner

University of Washington

{lerner, aksimpso, yoshi, franzi}@cs.washington.edu

Abstract

Though web tracking and its privacy implications have received much attention in recent years, that attention has come relatively recently in the history of the web and lacks full historical context. In this paper, we present longitudinal measurements of third-party web tracking behaviors from 1996 to present (2016). Our tool, *TrackingExcavator*, leverages a key insight: that the Internet Archive’s Wayback Machine opens the possibility for a retrospective analysis of tracking over time. We contribute an evaluation of the Wayback Machine’s view of past third-party requests, which we find is imperfect—we evaluate its limitations and unearth lessons and strategies for overcoming them. Applying these strategies in our measurements, we discover (among other findings) that third-party tracking on the web has increased in prevalence and complexity since the first third-party tracker that we observe in 1996, and we see the spread of the most popular trackers to an increasing percentage of the most popular sites on the web. We argue that an understanding of the ecosystem’s historical trends—which we provide for the first time at this scale in our work—is important to any technical and policy discussions surrounding tracking.

1 Introduction

Third-party web tracking is the practice by which third parties like advertisers, social media widgets, and website analytics engines—embedded in the first party sites that users visit directly—re-identify users across domains as they browse the web. Web tracking, and the associated privacy concerns from tracking companies building a list of sites users have browsed to, has inspired a significant and growing body of academic work in the computer security and privacy community, attempting to understand, measure, and defend against such tracking (e.g., [3, 4, 6, 8, 14, 15, 18–20, 22, 24, 25, 27–30, 32–34, 37, 39–43, 45, 46, 51, 57, 60, 61, 64–66, 70, 71]).

However, the research community’s interest in web tracking comes relatively recently in the history of web. To our knowledge, the earliest measurement studies began in 2005 [42], with most coming after 2009—while display advertising and the HTTP cookie standard date to the mid-1990s [44, 48]. Though numerous studies have now been done, they typically consist of short-term measurements of specific tracking techniques. We argue that public and private discussions surrounding web tracking—happening in technical, legal, and policy arenas (e.g., [49, 72])—ought to be informed not just by a single snapshot of the web tracking ecosystem but by a comprehensive knowledge of its trajectory over time. We provide such a comprehensive view in this paper, conducting a measurement study of third-party web tracking across 20 years since 1996.

Measurement studies of web tracking are critical to provide transparency for users, technologists, policy-makers, and even those sites that include trackers, to help them understand how user data is collected and used, to enable informed decisions about privacy, and to incentivize companies to consider privacy. However, the web tracking ecosystem is continuously evolving, and others have shown that web privacy studies at a single point in time may only temporarily reduce the use of specific controversial tracking techniques [63]. While one can study tracking longitudinally starting in the present, as we and others have (e.g., [42, 63]), ideally any future developments in the web tracking ecosystem can be contextualized in a comprehensive view of that ecosystem over time—i.e., since the very earliest instance of tracking on the web. We provide that longitudinal, historical context in this paper, asking: how has the third-party web tracking ecosystem evolved since its beginnings?

To answer this question, we apply a key insight: the Internet Archive’s *Wayback Machine* [31] enables a retrospective analysis of third-party tracking on the web

*Co-first authors listed in alphabetical order.

over time. The Wayback Machine¹ contains archives of full webpages, including JavaScript, stylesheets, and embedded resources, dating back to 1996. To leverage this archive, we design and implement a retrospective tracking detection and analysis platform called *TrackingExcavator* (Section 3), which allows us to conduct a longitudinal study of third-party tracking from 1996 to present (2016). *TrackingExcavator* logs in-browser behaviors related to web tracking, including: third-party requests, cookies attached to requests, cookies programmatically set by JavaScript, and the use of other relevant JavaScript APIs (e.g., HTML5 LocalStorage and APIs used in browser fingerprinting [15, 57], such as enumerating installed plugins). *TrackingExcavator* can run on both live as well as archived versions of websites.

Harnessing the power of the Wayback Machine for our analysis turns out to be surprisingly challenging (Section 4). Indeed, a key contribution of this paper is our evaluation of the historical data provided by the Wayback Machine, and a set of lessons and techniques for extracting information about trends in third-party content over time. Through a comparison with ground truth datasets collected in 2011 (provided to us by the authors of [60]), 2013, 2015, and 2016, we find that the Wayback Machine’s view of the past, as it relates to included third-party content, is imperfect for many reasons, including sites that were not archived due to `robots.txt` restrictions (which are respected by the Wayback Machine’s crawlers), the Wayback Machine’s occasional failure to archive embedded content, as well as site resources that were archived at different times than the top-level site. Though popular sites are typically archived at regular intervals, their embedded content (including third-party trackers) may thus be only partially represented. Whereas others have observed similar limitations with the Wayback Machine, especially as it relates to content visible on the top-level page [10, 38, 53], our analysis is focused on the technical impact of missing third-party elements, particularly with respect to tracking. Through our evaluation, we characterize what the Wayback Machine lets us measure about the embedded third parties, and showcase some techniques for best using the data it provides and working around some of its weaknesses (Section 4).

After evaluating the Wayback Machine’s view into the past and developing best practices for using its data, we use *TrackingExcavator* to conduct a longitudinal study of the third-party web tracking ecosystem from 1996–2016 (Sections 5). We explore how this ecosystem has changed over time, including the prevalence of different web tracking behaviors, the identities and scope of popular trackers, and the complexity of relationships within

the ecosystem. Among our findings, we identify the earliest tracker in our dataset in 1996 and observe the rise and fall of important players in the ecosystem (e.g., the rise of Google Analytics to appear on over a third of all popular websites). We find that websites contact an increasing number of third parties over time (about 5% of the 500 most popular sites contacted at least 5 separate third parties in early 2000s, whereas nearly 40% do so in 2016) and that the top trackers can track users across an increasing percentage of the web’s most popular sites. We also find that tracking behaviors changed over time, e.g., that third-party popups peaked in the mid-2000s and that the fraction of trackers that rely on referrals from other trackers has recently risen.

Taken together, our findings show that third-party web tracking is a rapidly growing practice in an increasingly complex ecosystem—suggesting that users’ and policy-makers’ concerns about privacy require sustained, and perhaps increasing, attention. Our results provide hitherto unavailable historical context for today’s technical and policy discussions.

In summary, our contributions are:

1. *TrackingExcavator*, a **measurement infrastructure** for detecting and analyzing third-party web tracking behaviors in the present and—leveraging the Wayback Machine—in the past (Section 3).
2. An **in-depth analysis** of the scope and accuracy of the Wayback Machine’s view of historical web tracking behaviors and trends, and techniques for working around its weaknesses (Section 4).
3. A **longitudinal measurement study** of third-party cookie-based web tracking from 1996 to present (2016)—to the best of our knowledge, the longest longitudinal study of tracking to date (Section 5).

This paper and any updates, including any data or code we publish, will be made available at <http://trackingexcavator.cs.washington.edu/>.

2 Background and Motivation

Third-party web tracking is the practice by which entities (“trackers”) embedded in webpages re-identify users as they browse the web, collecting information about the websites that they visit [50, 60]. Tracking is typically done for the purposes of website analytics, targeted advertising, and other forms of personalization (e.g., social media content). For example, when a user visits `www.cnn.com`, the browser may make additional requests to `doubleclick.net` to load targeted ads and to `facebook.com` to load the “Like” button; as a result, Doubleclick and Facebook learn about that user’s visit to CNN. Cookie-based trackers re-identify users by setting unique identifiers in browser cookies, which are then automatically included with requests to the tracker’s domain. Figure 1 shows a basic example; we discuss

¹<https://archive.org>

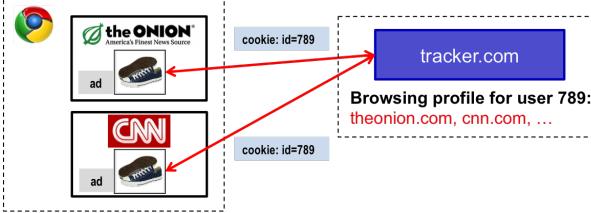


Figure 1: Overview of basic cookie-based web tracking. The third-party domain `tracker.com` uses a browser cookie to re-identify users on sites that embed content from `tracker.com`. This example shows *vanilla* tracking according to the taxonomy from [60]; other behaviors are described in Section 3.

more complex cookie-based tracking behaviors in Section 3. Though cookie-based tracking is extremely common [60], other types of tracking behaviors have also emerged, including the use of other client-side storage mechanisms, such as HTML5 LocalStorage, or the use of browser and/or machine fingerprinting to re-identify users without the need to store local state [15, 57].

Because these embedded trackers are often invisible to users and not visited intentionally, there has been growing concern about the privacy implications of third-party tracking. In recent years, it has been the subject of repeated policy discussions (Mayer and Mitchell provide an overview as of 2012 [50]); simultaneously, the computer science research community has studied tracking mechanisms (e.g., [50, 57, 60, 71]), measured their prevalence (e.g., [3, 20, 42, 60]), and developed new defenses or privacy-preserving alternatives (e.g., [6, 22, 25, 61, 64]). We discuss related works further in Section 6.

However, the research community’s interest in web tracking is relatively recent, with the earliest measurements (to our knowledge) beginning in 2005 [42], and each study using a different methodology and measuring a different subset of known tracking techniques (see Englehardt et al. [18] for a comprehensive list of such studies). The practices of embedding third-party content and targeted advertising on websites predate these first studies [48], and longitudinal studies have been limited. However, longitudinal studies are critical to ensure the sustained effects of transparency [63] and to contextualize future measurements. Thus, to help ground technical and policy discussions surrounding web tracking in historical trends, we ask: how has the third-party tracking ecosystem evolved over the lifetime of the web?

We investigate questions such as:

- How have the **numbers, identities, and behaviors** of dominant trackers changed over time?
- How has the **scope** of the most popular trackers (i.e., the number of websites on which they are embedded) changed over time?
- How has the **prevalence** of tracking changed over time? For example, do websites include many more

third-party trackers now than they did in the past?

- How have the **behaviors** of web trackers (e.g., JavaScript APIs used) changed over time?

By answering these questions, we are able to provide a systematic and longitudinal view of third-party web tracking over the last 20 years, retroactively filling this gap in the research literature, shedding a light on the evolution of third-party tracking practices on the web, and informing future technical and policy discussions.

The Wayback Machine. To conduct our archeological study, we rely on data from the Internet Archive’s Wayback Machine (<https://archive.org>). Since 1996, the Wayback Machine has archived full webpages, including JavaScript, stylesheets, and any resources (including third-party JavaScript) that it can identify statically from the site contents. It mirrors past snapshots of these webpages on its own servers; visitors to the archive see the pages as they appeared in the past, make requests for all resources from the Wayback Machine’s archived copy, and execute all JavaScript that was archived. We evaluate the completeness of the archive, particularly with respect to third-party requests, in Section 4.

3 Measurement Infrastructure: TrackingExcavator

To conduct a longitudinal study of web tracking using historical data from the Wayback Machine, we built a tool, TrackingExcavator, with the capability to (1) detect and analyze third-party tracking-related behaviors on a given web page, and (2) run that analysis over historical web pages archived and accessed by the Wayback Machine. In this section, we introduce TrackingExcavator. Figure 2 provides an overview of TrackingExcavator, which is organized into four pipeline stages:

(1) Input Generation (Section 3.1): TrackingExcavator takes as input a list of top-level sites on which to measure tracking behaviors (such as the Alexa top 500 sites), and, in “Wayback mode,” a timestamp for the desired archival time to create `archive.org` URLs.

(2) Data Collection (Section 3.2): TrackingExcavator includes a Chrome browser extension that automatically visits the pages from the input set and collects tracking-relevant data, such as third-party requests, cookies, and the use of certain JavaScript APIs.

(3) Data Analysis (Section 3.3): TrackingExcavator processes collected measurement events to detect and categorize third-party web tracking behaviors.

(4) Data Visualization: Finally, we process our results into visual representations (included in Section 5).

3.1 Input Generation

In the input generation phase, we provide TrackingExcavator with a list of top-level sites to use for measurement.

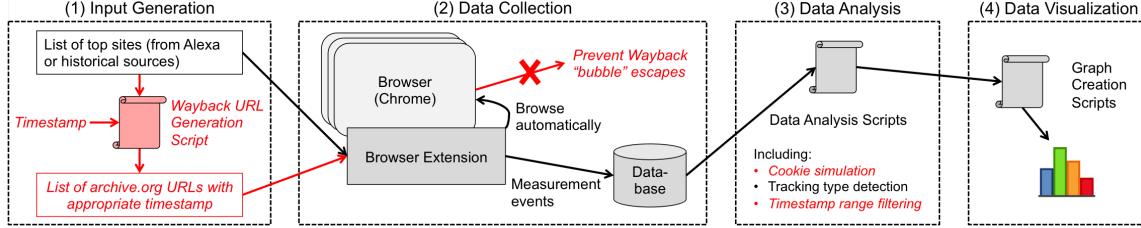


Figure 2: Overview of our infrastructure, TrackingExcavator, organized into four pipeline stages. Red/italic elements apply only to “Wayback mode” for historical measurements, while black/non-italics elements apply also to present-day measurements.

For historical measurements, TrackingExcavator must take a list of top-level URLs along with historical timestamps and transform them into appropriate URLs on archive.org. For example, the URL for the Wayback Machine’s February 10, 2016 snapshot of <https://www.usenix.org/conference/usenixsecurity16> is <https://web.archive.org/web/20160210050636/><https://www.usenix.org/conference/usenixsecurity16>.

We use the Memento API to find the nearest archived snapshot of a website occurring before the specified measurement date [36]. Though this process ensures a reasonable timestamp for the top-level page, embedded resources may have been archived at different times [5]. During analysis, we thus filter out archived resources whose timestamps are more than six months from our measurement timestamp, to ensure minimal overlap and sufficient spacing between measurements of different years.

3.2 Data Collection

To collect data, TrackingExcavator uses a Chrome extension to automatically visit the set of input sites. Note that we cannot log into sites, since the Wayback Machine cannot act as the original server. Our browser is configured to allow third-party cookies as well as pop-ups, and we visit the set of sites twice: once to prime the cache and the cookie store (to avoid artifacts of first-time browser use), and once for data collection. During these visits, we collect the following information relevant to third-party web tracking and store it in a local database:

- All request and response headers (including `set-cookie`).
- All cookies programmatically set by JavaScript (using `document.cookie`).
- All accesses to fingerprint-related JavaScript APIs, as described below.
- For each request: the requested URL, (if available) the referrer, and (if available) information about the originating tab, frame, and window.

We later process this data in the analysis phase of TrackingExcavator’s pipeline (Section 3.3 below).

Fingerprint-Related APIs. Since cookie-based web tracking is extremely common (i.e., it is “classic” web tracking), we focus largely on it—and third-party requests in general—to capture the broadest view of the web tracking ecosystem over time. However, we also collect information about the uses of other, more recently emerged tracking-related behaviors, such as JavaScript APIs that may be used to create browser or machine fingerprints [15, 57]. To capture any accesses a webpage makes to a fingerprint-related JavaScript API (such as `navigator.userAgent`), TrackingExcavator’s Chrome extension Content Script overwrites these APIs on each webpage to (1) log the use of that API and (2) call the original, overwritten function. The set of APIs that we hook was collected from prior work on fingerprint-based tracking [3, 4, 15, 56, 57] and is provided in Appendix A.

Preventing Wayback “Escapes”. In archiving a page, the Wayback Machine transforms all embedded URLs to archived versions of those URLs (similar to our own process above). However, sometimes the Wayback Machine fails to properly identify and rewrite embedded URLs. As a result, when that archived page is loaded on archive.org, some requests may “escape” the archive and reference resources on the live web [9, 38]. In our data collection phase, we block such requests to the live web to avoid anachronistic side effects. However, we record the domain to which such a request was attempted, since the archived site did originally make that request, and thus we include it in our analysis.

3.3 Data Analysis

In designing TrackingExcavator, we chose to separate data collection from data analysis, rather than detecting and measuring tracking behaviors on the fly. This modular architecture simplifies data collection and isolates it from possible bugs or changes in the analysis pipeline—allowing us to rerun different analyses on previously collected data (e.g., to retroactively omit certain domains).

“Replaying” Events. Our analysis metaphorically “replays” collected events to simulate loading each page in the measurement. For historical measurements, we modify request headers to replace “live web” Set-Cookie headers with X-Archive-Orig-Set-Cookie headers

added by `archive.org`, stripping the Wayback Machine prefixes from request and referrer URLs, and filling our simulated cookie jar (described further below). During the replay, TrackingExcavator analyzes each event for tracking behaviors.

Classifying Tracking Behaviors. For *cookie-based trackers*, we base our analysis on a previously published taxonomy [60].² We summarize—and augment—that taxonomy here. Note that a tracker may fall into multiple categories, and that a single tracker may exhibit different behaviors across different sites or page loads:

1. *Analytics Tracking*: The tracker provides a script that implements website analytics functionality. Analytics trackers are characterized by a script, sourced from a third party but run in the first-party context, that sets first-party cookies and later leaks those cookies to the third-party domain.
2. *Vanilla Tracking*: The tracker is included as a third party (e.g., an iframe) in the top-level page and uses third-party cookies to track users across sites.
3. *Forced Tracking*: The tracker forces users to visit its domain directly—for example, by opening a popup or redirecting the user to a full-page ad—allowing it to set cookies from a first-party position.
4. *Referred Tracking*: The tracker relies on another tracker to leak unique identifiers to it, rather than on its own cookies. In a hypothetical example, `adnetwork.com` might set its own cookie, and then explicitly leak that cookie in requests to referred tracker `ads.com`. In this case, `ads.com` need not set its own cookies to perform tracking.
5. *Personal Tracking*: The tracker behaves like a Vanilla tracker but is visited by the user directly in other contexts. Personal trackers commonly appear as social widgets (e.g., “Like” or “tweet” buttons).

In addition to these categories previously introduced [60], we discovered an additional type of tracker related to but subtly different from Analytics tracking:

6. *Referred Analytics Tracking*: Similar to an Analytics tracker, but the domain which sets a first-party cookie is *different* from the domain to which the first-party cookie is later leaked.

Beyond cookie-based tracking behaviors, we also consider the use of fingerprint-related JavaScript APIs, as described above. Though the use of these APIs does not necessarily imply that the caller is fingerprinting the user—we know of no published heuristic for determining fingerprinting automatically—but the use of many such APIs may suggest *fingerprint-based tracking*.

Finally, in our measurements we also consider *third-party requests* that are not otherwise classified as track-

²We are not aware of other taxonomies of this granularity for cookie-based tracking.

ers. If contacted by multiple domains, these third-parties have the *ability* to track users across sites, but may or may not actually do so. In other words, the set of all domains to which we observe a third-party request provides an upper bound on the set of third-party trackers.

We tested TrackingExcavator’s detection and classification algorithms using a set of test websites that we constructed and archived using the Wayback Machine, triggering each of these tracking behaviors.

Reconstructing Archived Cookies. For many tracking types, the presence or absence of cookies is a key factor in determining whether the request represents a tracking behavior. In our live measurements, we have the actual `Cookie` headers attached by Chrome during the crawl. On archived pages, the Wayback Machine includes past `Set-Cookie` headers as `X-Archive-Orig-Set-Cookie` headers on archived responses. To capture the cookies that would have actually been set during a live visit to that archived page, TrackingExcavator must simulate a browser cookie store based on these archival cookie headers and JavaScript cookie set events recorded during data collection.

Unfortunately, cookie engines are complicated and standards non-compliant in major browsers, including Chrome [11]. Python’s cookie storage implementation is compliant with RFC 2965, obsoleted by RFC 6265, but these standards proposals do not accurately represent modern browser practices [7, 13, 21]. For efficiency, we nevertheless use Python’s cookie jar rather than attempting to re-implement Chrome’s cookie engine ourselves.

We found that Python’s cookie jar computed cookies exactly matching Chrome’s for only 71% of requests seen in a live run of the top 100. However, for most types of tracking, we only need to know whether *any* cookies would have been set for the request, which we correctly determine 96% of the time. Thus our tool captures most tracking despite using Python’s cookie jar.

Classifying Personal Trackers in Measurements. For most tracker types, classification is independent of user behaviors. Personal trackers, however, are distinguished from Vanilla trackers based on whether the user visits that domain as a top-level page (e.g., Facebook or Google). To identify likely Personal trackers in automated measurement, we thus develop a heuristic for user browsing behaviors: we use popular sites from each year, as these are (by definition) sites that many users visited.

Alexa’s top sites include several that users would not typically visit directly, e.g., `googleadservices.com`. Thus, we manually examined lists of popular sites for each year to distinguish between domains that users *typically* visit intentionally (e.g., Facebook, Amazon) from those which ordinary users never or rarely visit intentionally (e.g., ad networks or CDNs). Two researchers

independently classified the domains on the Alexa top 100 sites for each year where we have Alexa data, gathering information about sites for which they were unsure. The researchers examined 435 total domains: for the top 100 domains in 2015, they agreed on 100% and identified 94 sites as potential Personal trackers; for the 335 additional domains in the previous years’ lists, they agreed on 95.4% and identified 296 Personal tracker domains.

4 Evaluating the Wayback Machine as an Archaeological Data Source for Tracking

The Wayback Machine provides a unique and comprehensive source of historical web data. However, it was not created for the purpose of studying third-party web tracking and is thus imperfect for that use. Nevertheless, the only way to study web tracking *prior* to explicit measurements targeting it is to leverage materials previously archived for other purposes. Therefore, before using the Wayback Machine’s archived data, it is essential to systematically characterize and analyze its capabilities and flaws in the context of third-party tracking.

In this section we thus study the extent to which data from the Wayback Machine allows us to study historical web tracking behaviors. Beyond providing confidence in the trends of web tracking over time that we present in Section 5, we view this evaluation of the Wayback Machine as a contribution of this paper. While others have studied the quality of the Wayback Machine’s archive, particularly with respect to the quality of the archived content displayed on the top-level page (e.g., [10, 38, 53]), we are the first to systematically study the quality of the Wayback Machine’s data about *third-party requests*, the key component of web tracking.

To conduct our evaluation, we leverage four ground truth data sets collected from the live web in 2011, 2013, 2015, and 2016. The 2011 data was originally used in [60] and provided to us by those authors. All datasets contain classifications of third-party cookie-based trackers (according to the above taxonomy) appearing on the Alexa top 500 sites (from the time of each measurement). The 2015 and 2016 data was collected by TrackingExcavator and further contains all HTTP requests, including those not classified as tracking.³ We plan to release our ground truth datasets from 2013, 2015, and 2016.

We organize this section around a set of lessons that we draw from this evaluation. We apply these lessons in our measurements in Section 5. We believe our findings can assist future researchers seeking to use the Wayback Machine as a resource for studying tracking (or other web properties relying on third-party requests) over time.

³For comparison, the published results based on the 2011 dataset [60] measured tracking on the homepages of the top 500 websites as well as four additional pages on that domain; for the purposes of our work, we re-analyzed the 2011 data using only homepages.

	August 1	August 25	September 1
All Third-Parties	324	304	301
Analytics	7	13	11
Vanilla	127	115	108
Forced	0	0	0
Referred	3	3	3
Personal	23	21	21
Referred Analytics	21	17	18

Table 1: Natural variability in the trackers observed on different visits to the Alexa top 100 in 2015. This variability can result from non-static webpage content, e.g., ad auctions that result in different winners.

4.1 Lesson (Challenge): The Wayback Machine provides a partial view of third-party requests

A key question for using the Wayback Machine for historical measurements is: how complete is the archive’s view of the past, both for the top-level pages and for the embedded content? In this lesson, we explore why its view is incomplete, surfacing challenges that we will overcome in subsequent lessons. We identify several reasons for the differences between the live and Wayback measurements, and quantify the effects of each.

Variation Between Visits. Different trackers and other third parties may appear on a site when it is loaded a second time, even if these views are close together; an example of this variation would be disparity in tracking behaviors between ads in an ad network.

To estimate the degree of variation between page views, we compare three live runs from August–September 2015 of the Alexa top 100 sites (Table 1). We find that variation between runs even a week apart is notable (though not enough to account for all of the differences between Wayback and live datasets). For the number of Vanilla trackers found, the August 25th and September 1st runs vary by 7 trackers, or 6%.

Non-Archived and Blocked Requests. There are several reasons that the Wayback Machine may fail to archive a response to a request, or provide a response that TrackingExcavator must ignore (e.g., from a far different time than the one we requested or from the live web). We describe these conditions here, and evaluate them in the context of a Wayback Machine crawl of the top 500 pages archived in 2015, according to the 2015 Alexa top 500 rankings; we elaborate on this dataset in Section 5. Table 2 summarizes how often the various conditions occur in this dataset, for requests, unique URLs, and unique domains. In the case of domains, we count only those domains for which *all* requests are affected, since those are the cases where we will *never* see a cookie or any other subsequent tracking indicators for that domain.

Robots.txt Exclusions (403 errors). If a domain’s robots.txt asks that it not be crawled, the Wayback Machine will respect that restriction and thus not archive

Type of Blocking	Fraction Missed
Robots Exclusions	Requests 1115 / 56,173 (2.0%)
	URLs 609 / 27,532 (2.2%)
	Domains 18 / 1150 (1.6%)
Not Archived	Requests 809 / 56,173 (1.4%)
	URLs 579 / 27,532 (2.1%)
	Domains 8 / 1150 (0.7%)
Wayback Escapes	Requests 9025 / 56,173 (16.1%)
	URLs 4730 / 27,532 (17.2%)
	Domains 132 / 1150 (11.5%)
Inconsistent Timestamps	Requests 404 / 56,173 (0.7%)
	URLs 156 / 27,532 (0.6%)
	Domains 55 / 1150 (4.8%)

Table 2: For the archived versions of the Alexa top 500 sites from 2015, the fraction of requests, unique URLs, and unique domains affected by robots exclusion (403 errors), not archived (404), Wayback escapes (blocked by TrackingExcavator), or inconsistent timestamps (filtered by TrackingExcavator).

the response. As a result, we will not receive any information about that site (including cookies, or use of Javascript) nor will we see any subsequent resources that would have resulted from that request.

We find that only a small fraction of all requests, unique URLs, and (complete) domains are affected by robots exclusion (Table 2). We note that robots exclusions are particularly common for popular trackers. Of the 20 most popular trackers on the 2015 live dataset, 12 (60%) are blocked at least once by robots.txt in the 2015 Wayback measurement. By contrast, this is true for only 74/456, or 16.23%, of all Vanilla trackers seen in live.

Other Failures to Archive (404 errors). The Wayback Machine may fail to archive resources for any number of reasons. For example, the domain serving a certain resource may have been unavailable at the time of the archive, or changes in the Wayback Machine’s crawler may result in different archiving behaviors over time. As shown in Table 2, missing archives are rare.

URL Rewriting Failures (Wayback “Escapes”). Though the Wayback Machine’s archived pages execute the corresponding archived JavaScript within the browser when TrackingExcavator visits them, the Wayback Machine does not execute JavaScript during its archival crawls of the web. Instead, it attempts to statically extract URLs from HTML and JavaScript to find additional sites to archive. It then modifies the archived JavaScript, rewriting the URLs in the included script to point to the archived copy of the resource. This process may fail, particularly for dynamically generated URLs. As a result, when TrackingExcavator visits archived pages, dynamically generated URLs not properly redirected to their archived versions will cause the page to attempt to make a request to the live web, i.e., “escape” the archive. TrackingExcavator blocks such escapes (see Section 3). As a result, the script never runs on the archived site, never sets a cookie or leaks it, and thus TrackingExcavator

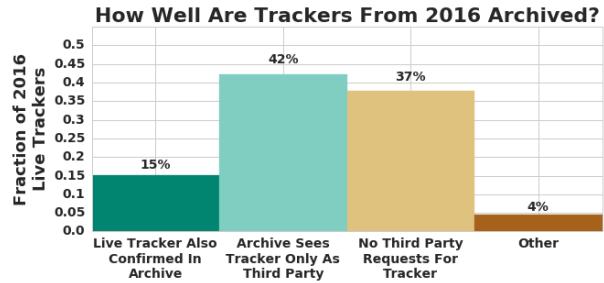


Figure 3: The fraction of domains categorized as Vanilla trackers in the live 2015 crawl which, in the archival 2015 crawl, (1) set and leaked cookies and thus were confirmed as trackers, (2) were only third-party requests (had at least one third-party request but no cookies), (3) did not appear at all, or (4) other (e.g., had cookies but not at the time of a third-party request, or cookies were not attached due to a cookie simulation bug).

tor does not witness the associated tracking behavior.

We find that Wayback “escapes” are more common than robots exclusion or missing archives (Table 2): 16.1% of all requests attempted to “escape” (i.e., were not properly rewritten by the Wayback Machine) and were blocked by TrackingExcavator.

Inconsistent Timestamps. As others have documented [10], embedded resources in a webpage archived by the Wayback Machine may occasionally have a timestamp far from the timestamp of the top-level page. As described in Section 3, we ignore responses to requests for resources with timestamps more than six months away.

Cascading Failures. Any of the above failures can lead to cascading failures, in that non-archived responses or blocked requests will result in the omission of any subsequent requests or cookie setting events that would have resulted from the success of the original request. The “wake” of a single failure cannot be measured within an archival dataset, because events following that failure are simply missing. To study the effect of these cascading failures, we must compare an archival run to a live run from the same time; we do so in the next subsection.

4.2 Lesson (Opportunity): Consider all third-party requests, in addition to confirmed trackers

In the previous section, we evaluated the Wayback Machine’s view of third-party requests *within* an archival measurement. For requests affected by the issues in Table 2, TrackingExcavator observes the existence of these requests — i.e., counts them as third parties — but without the corresponding response may miss additional information (e.g., set cookies) that would allow it to confirm these domains as trackers according to the taxonomy presented earlier. However, this analysis cannot give us a sense of how many third-party requests are entirely absent from Wayback data due to cascading failures, nor a sense of any other data missing from the archive, such as

	2011	2013	2015	2016
Wayback (All Third Parties)	553	621	749	723
Wayback (Vanilla+Personal)	47	49	92	90
Live (Vanilla+Personal)	370	419	493	459
Wayback-to-Live Ratio (Vanilla+Personal)	0.13	0.12	0.19	0.20

Table 3: We compare the prevalence of the most common tracking types (Vanilla and Personal) over the four years for which we have data from the live web. Though the Wayback Machine provides only partial data on trackers, it nevertheless illuminates a general upward trend reflected in our ground truth data.

missing cookie headers on otherwise archived responses. For that, we must compare directly with live results.

We focus our attention on unique trackers: we attempt to identify which live trackers are missing in the 2015 Wayback dataset, and why. For each tracker we observe in our 2015 live measurement, Figure 3 identifies whether we (1) also observe that tracker in “Wayback mode,” (2) observe only a third-party request (but no confirmed cookie-based tracking behavior, i.e., we classify it only as a third-party domain), or (3) do not observe any requests to that tracker at all.

We conclude two things from this analysis. First, because the Wayback Machine may fail to provide sufficient data about responses or miss cookies even in archived responses, many trackers confirmed in the live dataset appear as simple third-party requests in the Wayback data (the second column in Figure 3). For example, `doubleclick.net`, one of the most popular trackers, appears as only a third party in Wayback data because of its `robots.txt` file. Thus, we learn that to study third-party web tracking in the past, due to missing data in the archive, *we must consider all third-party requests*, not only those confirmed as trackers according to the taxonomy. Though considering only third-party requests will overcount tracking in general (i.e., not all third parties on the web are trackers), we find that it broadens our view of tracking behaviors in the archive.

Second, we find that a non-trivial fraction of trackers are missing entirely from the archive (the third column in Figure 3). In the next subsection, we show that we can nevertheless draw conclusions about trends over time, despite the fact that the Wayback Machine under-represents the raw number of third parties contacted.

4.3 Lesson (Opportunity): The Wayback Machine’s data allows us to study trends over time

As revealed above, the Wayback Machine’s view of the past may miss the presence of some third parties entirely. Thus, one unfortunately cannot rely on the archive to shed light on the exact raw numbers of trackers and other third parties over time. Instead, we ask: does the Wayback Machine’s data reveal genuine historical *trends*?

To investigate trends, we compare all of our live

datasets (2011, 2013, 2015, and 2016) to their Wayback counterparts. Table 3 compares the number of Vanilla and Personal trackers (the most prevalent types) detected in each dataset. For the purposes of this comparison, we sum the two types, since their distinction depends only on the user’s browsing behaviors. We also include the number of all third parties in the Wayback datasets, based on the previous lesson. Though not all of these third parties represent trackers in live data, they help illuminate trends in third party prevalence over time.

We draw two conclusions from this comparison. First, we find that we *can* rely on the archive to illuminate general trends over time. Although confirmed trackers in “Wayback mode” (as expected from our earlier lessons) underrepresent the number of confirmed trackers found on the live web—and third parties in the archive overestimate confirmed trackers in the live data—we find that the trends we see over time are comparable in both sets of measurements. Critically, we see that *the upward trend in our archival view is not merely the result of improvements in archive quality over time or other factors—we indeed observe this trend reflected in ground truth data*. We gain further confidence in these trends in Section 5, where we see a rise in tracking behaviors since 1996 that corresponds with our intuition. The absence of any large vertical steps in the figures in Section 5 further suggests that the trends we identify are artifacts of the web evolving as opposed to any significant changes in the Wayback Machine archival process.

Second, however, we find that—although long-term trends appear to be meaningfully represented by the Wayback Machine—one should not place too much confidence into *small* variations in trends. For example, the Wayback Machine’s data in 2013 appears to be worse than in other years, under-representing the number of confirmed trackers more than average. Thus, in Section 5, we do not report on results that rely on small variations in trends unless we have other reasons to believe that these variations are meaningful.

4.4 Lesson (Opportunity): Popular trackers are represented in the Wayback Machine’s data

Because popular trackers, by definition, appear on many sites that users likely browse to, they have a strong effect on user privacy and are particularly important to examine. We find that although the Wayback Machine misses some trackers (for reasons discussed above), *it does capture a large fraction of the most popular trackers*—likely because the Wayback Machine is more likely to have correctly archived at least one of each popular tracker’s many appearances.

Specifically, when we examine the 2015 archival and live datasets, we find that 100% of the top 20 trackers from the live dataset are represented as either confirmed

trackers or other third parties in the Wayback data. In general, more popular trackers are better represented in Wayback data: 75% of the top 100 live trackers, compared to 53% of all live trackers. Tracker popularity drops quickly—the first live tracker missing in Wayback data is #22, which appears on only 22 of the top 500 websites; the 100th most popular tracker appears on only 4 sites. By contrast, the top tracker appears on 208 sites. In other words, those trackers that have the greatest impact on user privacy do appear in the archive.

Based on this lesson, we focus part of Section 5’s analysis on popular trackers, and we *manually label* those that the Wayback Machine only sees as third parties but that we know are confirmed trackers in live data.

4.5 Lesson (Opportunity): The Wayback Machine provides additional data beyond requests

Thus far, we have considered third-party requests and confirmed cookie-based trackers. However, the Wayback Machine provides, and TrackingExcavator collects, additional data related to web tracking behaviors, particularly the use of various JavaScript APIs that allow third parties to collect additional information about users and their machines (e.g., to re-identify users based on fingerprints). For JavaScript correctly archived by the Wayback Machine, TrackingExcavator observes accesses to the supported APIs (Appendix A). For example, we observe uses of `navigator.userAgent` as early as 1997.

4.6 Summary

In summary, we find that the Wayback Machine’s view of the past is incomplete, and that its weaknesses particularly affect the third-party requests critical for evaluating web tracking over time. We identified and quantified those weaknesses in Section 4.1, and then introduced findings and strategies for mitigating these weaknesses in Sections 4.2-4.5, including considering third-party requests as well as confirmed trackers, manually labeling known popular trackers, and studying general trends over time instead of raw numbers. We leverage these strategies in our own measurements. By surfacing and evaluating these lessons, we also intend to help guide future researchers relying on data from the Wayback Machine.

We focus on the Wayback Machine since it is to our knowledge the most comprehensive web archive. Applying our approach to other, more specialized archives [58], if relevant for other research goals, would necessitate a new evaluation of the form we presented here.

5 Historical Web Tracking Measurements

We now turn to our longitudinal study of third-party cookie-based web tracking from 1996-2016.

Datasets. We focus our investigation on the most popular websites each year, for two reasons: first, trackers

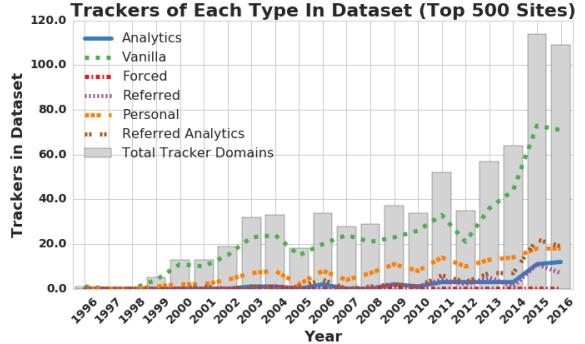


Figure 4: Evolution of tracker types over time. The grey bars show the total number of tracking domains present in each dataset, and the colored lines show the numbers of trackers with each type of tracking behavior. A single tracker may have more than one behavior in the dataset (e.g., both Vanilla and Analytics), so the sum of the lines might be greater than the bar.

on these sites are (or were) able to collect information about the greatest number of users; second, popular sites are crawled more frequently by the Wayback Machine (if permitted by `robots.txt`). We thus need historical lists of the top sites globally on the web.

2003-2016: Alexa. For 2010-2016, we use Wayback Machine archives of Alexa’s top million sites list (a csv file). For 2003-2009, we approximate the top 500 by scraping Alexa’s own historical API (when available) and archives of individual Alexa top 100 pages. Because of inconsistencies in those sources, our final lists contain 459-500 top sites for those years.

1996-2002: Popular Links from Homepages. In 2002, only the Alexa top 100 are available; before 2002, we only have ComScore’s list of 20 top sites [69]. Thus, to build a list of 500 popular sites for the years 1996-2002, we took advantage of the standard practice at the time of publishing links to popular domains on personal websites. Specifically, we located archives of the People pages of the Computer Science or similar department at the top 10 U.S. CS research universities as of 1999, as reported in that year by U.S. News Online [2]. We identified the top 500 domains linked to from the homepages accessible from those People pages, and added any ComScore domains that were not found by this process. We ran this process using People pages archived in 1996 and 1999; these personal pages were not updated or archived frequently enough to get finer granularity. We used the 1996 list as input to our 1996, 1997 and 1998 measurements, and the 1999 list as input for 1999-2002.

5.1 Prevalence of Tracking Behaviors over Time

We begin by studying the prevalence of tracking behaviors over time: how many unique trackers do we observe, what types of tracking behaviors do those trackers exhibit, and how many trackers appear on sites over time?

Prevalence and Behaviors of Unique Trackers. Figure 4 shows the total number of unique trackers observed over time (the grey bars) and the prevalence of different tracking behavior types (the lines) for the top 500 sites from 1996-2016. Note that trackers may exhibit more than one behavior across sites or on a single site, so the sum of the lines may be greater than the height of the bar. We note that the particularly large bars in 2015 and 2016 may reflect not only a change in tracking prevalence but also changes in the way the Wayback Machine archived the web. See Table 3 for validation against live data which suggest that actual growth may have been smaller and more linear, similar to past years.

We make several observations. First, we see the emergence of different tracking behaviors: the first cookie-based tracker in our data is from 1996: `microsoft.com` as a Vanilla tracker on `digital.net`. The first Personal tracker to appear in our dataset is in 1999: `go.com` shows up on 5 different sites that year, all also owned by Disney: `disney.com`, `espn.com`, `sportszone.com`, `wbs.net`, and `infoseek.com` (acquired by Disney mid-1999 [1], before the date of our measurement). The existence of a Personal tracker that only appeared on sites owned by the same company differs from today’s Personal tracking ecosystem, in which social media widgets like the Facebook “Like” button appear on many popular sites unaffiliated with that tracker (Facebook, in this case) [60].

More generally, we see a marked increase in quantities of trackers over time, with rises in all types of tracking behavior. One exception is Forced trackers — those relying on popups — which are rare and peaked in the early 2000s before popup blockers became default (e.g., in 2004 for Internet Explorer [54]). Indeed, we see third-party popups peak significantly in 2003 and 2004 (17 and 30 popups, respectively, compared to an annual mean of about 4), though we could not confirm all as trackers for Figure 4. Additionally, we see an increasing variety of tracking behavior over time, with early trackers nearly all simply Vanilla, but more recent rises in Personal, Analytics, and Referred tracking.

We can also consider the complexity of individual trackers, i.e., how many distinct tracking behaviors they exhibit over each year’s dataset. (Note that some behaviors are exclusive, e.g., a tracker cannot be both Personal and Vanilla, but others are nonexclusive.) Table 4 suggests that there has been some increase in complexity in recent years, with more trackers exhibiting two or even three behaviors. Much of this increase is due to the rise in Referred or Referred Analytics trackers, which receive cookie values shared explicitly by other trackers in addition to using their own cookies in Vanilla behavior.

Fingerprint-Related APIs. We measured the use of Javascript APIs which can be used to fingerprint

Year	1Type	2Type	3Type	4Type
1996	100.00% (1)	0	0	0
1998	0	0	0	0
2000	100.00% (13)	0	0	0
2002	100.00% (19)	0	0	0
2004	96.97% (32)	3.03% (1)	0	0
2006	100.00% (34)	0	0	0
2008	100.00% (29)	0	0	0
2010	94.12% (32)	2.94% (1)	2.94% (1)	0
2012	88.57% (31)	11.43% (4)	0	0
2014	93.75% (60)	4.69% (3)	1.56% (1)	0
2016	86.24% (94)	11.01% (12)	2.75% (3)	0

Table 4: Complexity of trackers, in terms of the percentage (and number) of trackers displaying one or more types of tracking behaviors across the top 500 sites.

Year	Most Prolific API-user	Num APIs Used	Coverage
1998	realhollywood.com	2	1
1999	go2net.com	2	1
2000	go.com	6	2
2001	akamai.net	8	15
2002	go.com	10	2
2003	bcentral.com	5	1
2004	163.com	9	3
2005	163.com	8	1
2006	sina.com.cn	11	2
2007	googlesyndication.com	8	24
2008	go.com	12	1
2009	clicksor.com	10	2
2010	tribalfusion.com	17	1
2011	tribalfusion.com	17	2
2012	imedia.cz	12	1
2013	imedia.cz	13	1
2014	imedia.cz	13	1
2015	aolcdn.com	25	5
2016	aolcdn.com	25	3

Table 5: Most prolific API-users, with ties broken by coverage (number of sites on which they appear) for each year. The maximum number of APIs used increases over time, but the max API users are not necessarily the most popular trackers.

browsers and persist identifiers even across cookie deletion. Though the use of these APIs does not necessarily imply that they are used for tracking (and we know of no published heuristic for correlating API use with genuine fingerprinting behaviors), the use of these APIs nevertheless allows third parties to gather potentially rich information about users and their machines. The full list of 37 fingerprint-related APIs we measure (based on prior work [3, 4, 15, 56, 57]) is in Appendix A.

We now consider third parties that are prolific users of fingerprint-related APIs, calling many APIs on each site. Table 5 shows the tracker in each year that calls the most APIs on a single site. Ties are broken by the choosing the third party that appears on the largest number of sites. Maximum usage of APIs has increased over time, but we observe that the most prolific API users are not the most popular cookie-based trackers. Although we only identify API uses within JavaScript, and not how their results are used, we note that increasing use

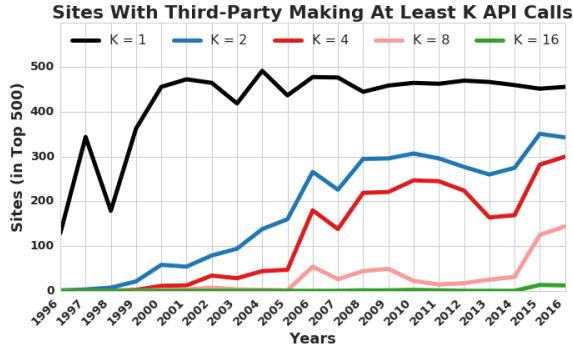


Figure 5: Number of sites in each year with a tracker that calls (on that site) at least K (of our 37) fingerprint-related APIs.

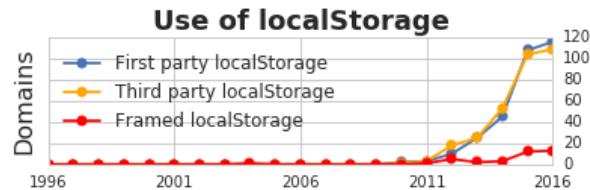


Figure 6: Domains using `window.localStorage`. First party usages are uses in the top frame of a web page by a script loaded from the web page’s own domain. Third party usages are those also in the top frame of a page but by a script loaded from a third party. Framed usages are those inside of an iframe.

of these APIs implies increased power to fingerprint, especially when combined with non-Javascript signals such as HTTP headers and plugin behavior. For example, Panopticlick derived 18 bits of entropy about remote browsers from a subset of these APIs plus HTTP headers and information from plugins [15].

Beyond the power of the most prolific fingerprint-related API users growing, we also find that more sites include more trackers using these APIs over time. Figure 5 shows the number of sites in each year containing a tracker that calls, on that site, at least K of the 37 fingerprinting APIs. Although many sites contain and have contained trackers that use at least 1 API (typically `navigator.userAgent`, common in browser compatibility checks), the number of sites containing trackers that call 2 or more APIs has risen significantly over time.

In addition to fingerprint-related APIs, we also examine the use of HTML5 LocalStorage, a per-site persistent storage mechanism standardized in 2009 in addition to cookies. Figure 6 shows that the use of the `localStorage` API rises rapidly since its introduction in 2009, indicating that tracking defenses should increasingly consider storage mechanisms beyond cookies.

Third Parties Contacted. We now turn our attention to the number of third parties that users encounter as they browse the web. Even third parties not confirmed as trackers have the *potential* to track users across the web, and as we discovered in Section 4, many third par-

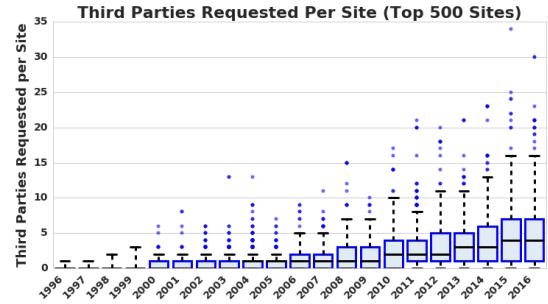


Figure 7: Distributions of third-party requests for the top 500 sites 1996–2016. Center box lines are medians, whiskers end at $1.5 \times \text{IQR}$. The increase in both medians and distributions of the data show that more third-parties are being contacted by popular sites in both the common and extreme cases.

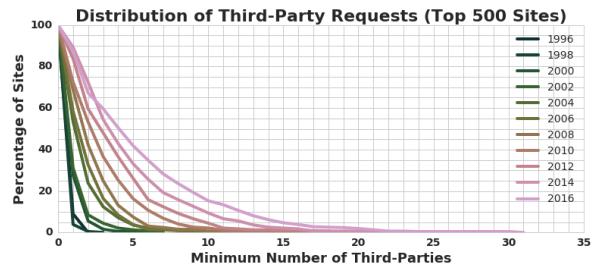


Figure 8: Distribution of top sites for each year by number of unique third-parties (tracking-capable domains) they contact. In later years, more sites appear to contact more third parties.

ties in archived data may in fact be confirmed trackers for which the Wayback Machine simply archived insufficient information. Figure 7 thus shows the distributions of how many third parties the top 500 sites contacted in each year. We see a rise in the median number of third parties contacted—in other words, more sites are giving more third parties the opportunity to track users.

Figure 8 provides a different view of similar data, showing the distribution of the top sites for each year by number of distinct third parties contacted. In the early 2000s, only about 5% of sites contacted at least 5 third parties, while in 2016 nearly 40% of sites did so. We see a maximum in 2015, when one site contacted 34 separate third-parties (a raw number that is likely underestimated by the Wayback Machine’s data)!

5.2 Top Trackers over Time

We now turn to an investigation of the top trackers each year: who are the top players in the ecosystem, and how wide is their view of users’ browsing behaviors?

Coverage of Top Trackers. We define the *coverage* of a set of trackers as the percentage of total sites from the dataset for which at least one of those trackers appears. For a single tracker, its coverage is the percentage of sites on which it appears. Intuitively, coverage suggests the concentration of tracking ability—greater coverage al-

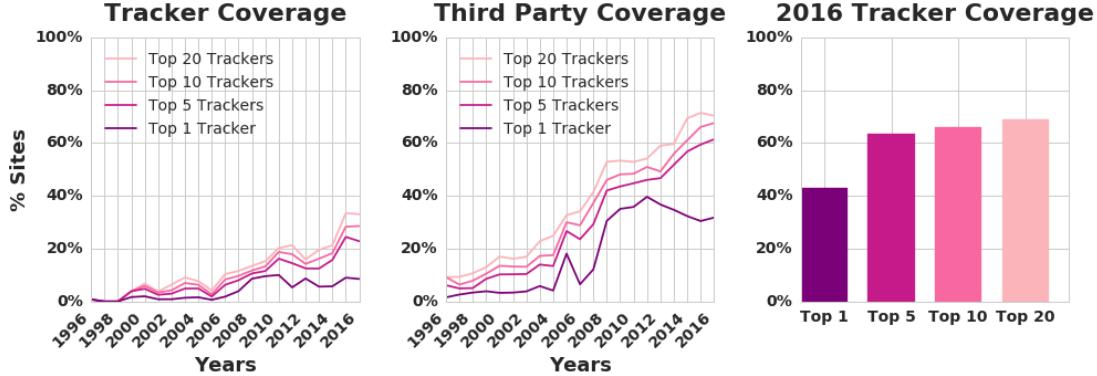


Figure 9: The growth in the coverage (percentage of top 500 sites tracked) of the top 1/5/10/20 trackers for each year is shown in the first and second panels, for all confirmed trackers and for all third parties respectively. The right hand panel shows the values on the live web for confirmed trackers, with the top 5 trackers covering about 70% of all sites in the dataset. Note that top third party coverage in the archive is an excellent proxy for modern confirmed tracker coverage today.

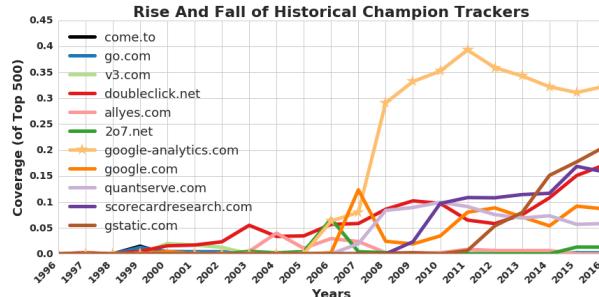


Figure 10: This figure depicts variations in site coverage for a number of the most popular confirmed trackers from years across the studied period. We call the two trackers embedded on the most sites in a given year the “champions” of that year, filtered by manual classification as described in the text.

lows trackers to build larger browsing profiles. This metric reaches toward the core privacy concern of tracking, that certain entities may know nearly everything a person does on the web. We consider trackers by domain name, even though some trackers are in fact owned by the same company (e.g., Google owns `google-analytics.com`, `doubleclick.net`, and the “+1” button served from `google.com`), because a business relationship does not imply that the entities share data, though some trackers may indeed share information out of public view.

Figure 9 illustrates the growth of tracker coverage over time. It considers both the single domain with the highest coverage for each year (Top 1 Tracker) as well as the combined coverage of the union of the top 5, 10 and 20 trackers. Confirming the lesson from Section 4.2, the coverage rates we see for third party domains in the archive are similar to live coverage of *confirmed* Vanilla cookie-based trackers.

Clearly, the coverage of top trackers has risen over time, suggesting that a small number of third parties can observe an increasing portion of user browsing histories.

Popular Trackers over Time. Who are these top track-

ers? Figure 10 shows the rise and fall of the top two trackers (“champions”) for each year. To create this figure, we make use of the lesson in Section 4.4 to manually label known popular confirmed trackers. We identified the two domains with the highest third-party request coverage for each year, omitting cases where the most popular tracker in a year appeared on only one site. We manually verified that 12/19 of these domains were in fact trackers by researching the domain, owning company, archived behavior and context, and modern behaviors (if applicable). Based on this analysis, we are able to assess the change in tracking behaviors even of domains for whom cookies are lost in the archive (e.g., `doubleclick.net`). In particular, this analysis reveals trends in the trackers with the most power to capture profiles of user behavior across many sites.

We find that in the early 2000s, no single tracker was present on more than 10% of top sites, but in recent years, `google-analytics.com` has been present on nearly a third of top sites and 2-4 others have been present on more than 10% and growing. Some, such as `doubleclick.net` (acquired by Google in 2008) have been popular throughout the entire time period of the graph, while others, such as `scorecardresearch.com`, have seen a much more recent rise.

We note that `google-analytics.com` is a remarkable outlier with nearly 35% coverage in 2011. Google Analytics is also an outlier in that it is one of only two non-cross-site trackers among the champions (`gstatic.com`, a Referred Analytics tracker, is the other). As an Analytics type tracker, Google Analytics tracks users only within a single site, meaning that its “coverage” is arguably less meaningful than that of a cross-site tracker. However, we observe that Google Analytics *could* track users across sites via fingerprinting or by changing its behavior to store tracking cookies. This observation highlights the need for repeated

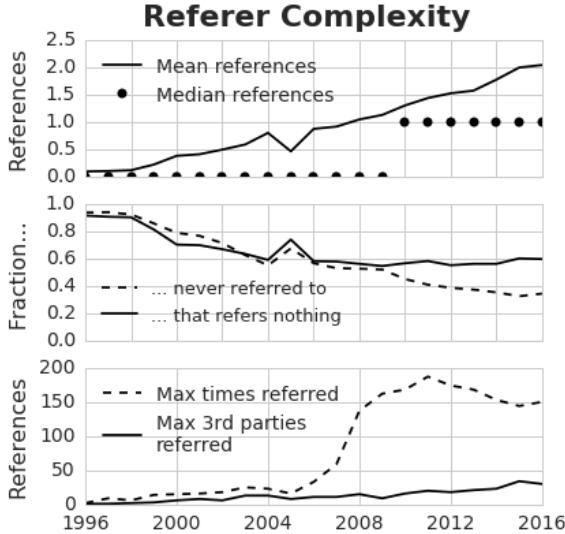


Figure 11: Changes in the frequency with which domains are referred to or refer to other domains (based on HTTP Referer).

measurements studies that provide transparency on the web: with a simple change to its tracking infrastructure, Google Analytics could begin to track users across 40% of the most popular sites on the web overnight. Thus, Google’s decision not to structure Google Analytics in this way has a tremendous impact on user privacy.

5.3 Evolution of the Tracking Ecosystem

Finally, we consider the tracking ecosystem as a whole, focusing on relationships between different trackers. We find a remarkable increase in the complexity of these relationships over time. Again we consider only relationships observable using TrackingExcavator, not external information about business relationships.

To study these relationships, we construct the graph of referring relationships between elements on pages. For example, if we observe a third-party request from `example.com` to `tracker.com`, or from `tracker.com` referring to `tracker2.com`, the nodes for those domains in the graph will be connected by edges.

We find a significant increase in complexity over time by examining several properties of this graph (Figure 11). Over time, the mean number of referrals outward from domains increases (top of Figure 11), while the number of domains that are never referred to by other domains or never refer outward steadily decreases (middle of Figure 11). Meanwhile, the maximum number of domains that refer to a single domain increases dramatically, suggesting that individual third parties in the web ecosystem have gradually gained increasing prominence and coverage. This reflects and confirms trends shown by other aspects of our data (Figures 10 and 9). These trends illuminate an ecosystem of generally increasingly connected relationships and players growing in size and influence.

Appendix B shows this evolution in graph form; the increase in complexity over time is quite striking.

5.4 Summary and Discussion

We have uncovered trends suggesting that tracking has become more prevalent and complex in the 20 years since 1996: there are now more unique trackers exhibiting more types of behaviors; websites contact increasing numbers of third parties, giving them the opportunity to track users; the scope of top trackers has increased, providing them with a broader view of user browsing behaviors; and the complexity and interconnectedness of the tracking ecosystem has increased markedly.

From a privacy perspective, our findings show that over time, more third parties are in a position to gather and utilize increasing amounts of information about users and their browsing behaviors. This increase comes despite recent academic, policy, and media attention on these privacy concerns and suggests that these discussions are far from resolved. As researchers continue to conduct longitudinal measurements of web tracking going forward, our work provides the necessary historical context in which to situate future developments.

6 Additional Related Work

Tracking and Defenses. Third-party tracking has been studied extensively in recent years, particularly through analysis and measurements from 2005 to present [18, 19, 24, 30, 32–34, 40–43, 60]. A few studies have considered mobile, rather than desktop, browser tracking [20, 27]. Beyond explicit stateful (e.g., cookie-based) tracking, recent work has studied the use of browser and machine fingerprinting techniques to re-identify and track users [3, 4, 15, 37, 57, 71]. Others have studied the possible results of tracking, including targeted ads [45, 70], personalized search [29], and price discrimination [66].

User-facing defenses against tracking range from browser extensions like Ghostery [23] and Privacy Badger [16] to research proposals (e.g. [8, 28]). Researchers have also designed privacy-preserving alternatives including privacy-preserving ads [22, 25, 59, 64], social media widgets [14, 39, 61], and analytics [6]. Others have studied user attitudes towards tracking and targeted advertising (e.g., [46, 51, 65]). Our study shows the increased prevalence of tracking over time, suggesting that designing and supporting these defenses for privacy-sensitive users is as important as ever.

Wayback Machine and other Longitudinal Measurements. Others have used the Wayback Machine for historical measurements to predict whether websites will become malicious [62] and to study JavaScript inclusion [55] and website accessibility [26]; to recover medical references [67]; to analyze social trends [35]; and as evidence in legal cases [17]. Others [53] found that

websites are accurately reflected in the archive. These studies noted similar limitations as we did, as well as ways it has changed over time [38]. Finally, researchers have studied other aspects of the web and Internet longitudinally without the use of archives, including IPv6 adoption [12], search-engine poisoning [47], privacy notices [52], and botnets [68].

7 Conclusion

Though third-party web tracking and its associated privacy concerns have received attention in recent years, the practice long predates the first academic measurements studies of tracking (begun in 2005). Indeed, in our measurements we find tracking behaviors as early as 1996. We introduce TrackingExcavator, a measurement infrastructure for third-party web tracking behaviors that leverages `archive.org`'s Wayback Machine to conduct historical studies. We rigorously evaluate the Wayback Machine's view of past third-party requests and develop strategies for overcoming its limitations.

We then use TrackingExcavator to conduct the most extensive longitudinal study of the third-party web tracking ecosystem to date, retrospectively from 1996 to present (2016). We find that the web tracking ecosystem has expanded in scope and complexity over time: today's users browsing the web's popular sites encounter more trackers, with more complex behaviors, with wider coverage, and with more connections to other trackers, than at any point in the past 20 years. We argue that understanding the trends in the web tracking ecosystem over time—provided for the first time at this scale by our work—is important to future discussions surrounding web tracking, both technical and political.

Beyond web tracking, there are many questions about the history and evolution of the web. We believe our evaluation of the Wayback Machine's view of the past, as well as TrackingExcavator, which we plan to release with this paper, will aid future study of these questions.

Acknowledgements

We thank individuals who generously offered their time and resources, and organizations and grants that support us and this work. Jason Howe of UW CSE offered invaluable technical help. Camille Cobb, Peter Ney, Will Scott, Lucy Simko, and Paul Vines read our drafts thoroughly and gave insightful feedback. We thank our colleagues from the UW Tech Policy Lab, particularly Ryan Calo and Emily McReynolds, for their thoughts and advice. This work was supported in part by NSF Grants CNS-0846065 and IIS-1302709, an NSF Graduate Research Fellowship under Grant No. DGE-1256082, and the Short-Dooley Professorship.

References

- [1] Disney absorbs Infoseek, July 1999. <http://money.cnn.com/1999/07/12/deals/disney/>.
- [2] Grad School Rankings, Engineering Specialties: Computer, 1999. <https://web.archive.org/web/19990427094034/http://www.usnews.com/usnews/edu/beyond/gradrank/gbengsp5.htm>.
- [3] ACAR, G., EUBANK, C., ENGLEHARDT, S., JUAREZ, M., NARAYANAN, A., AND DIAZ, C. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the ACM Conference on Computer and Communications Security* (2014).
- [4] ACAR, G., JUAREZ, M., NIKIFORAKIS, N., DIAZ, C., GÜRSES, S., PIJSSENS, F., AND PRENEEL, B. FP Detective: Dusting the web for fingerprints.
- [5] AINSWORTH, S. G., NELSON, M. L., AND VAN DE SOMPEN, H. Only One Out of Five Archived Web Pages Existed as Presented. 257–266.
- [6] AKKUS, I. E., CHEN, R., HARDT, M., FRANCIS, P., AND GEHRKE, J. Non-tracking web analytics. In *Proceedings of the ACM Conference on Computer and Communications Security* (2012).
- [7] BARTH, A. HTTP State Management Mechanism, Apr. 2011. <https://tools.ietf.org/html/rfc6265>.
- [8] BAU, J., MAYER, J., PASKOV, H., AND MITCHELL, J. C. A Promising Direction for Web Tracking Countermeasures. In *Web 2.0 Security and Privacy* (2013).
- [9] BRUNELLE, J. F. 2012-10-10: Zombies in the Archives. <http://ws-dl.blogspot.com/2012/10/2012-10-10-zombies-in-archives.html>.
- [10] BRUNELLE, J. F., KELLY, M., SALAHELDEN, H., WEIGLE, M. C., AND NELSON, M. L. Not All Mementos Are Created Equal : Measuring The Impact Of Missing Resources Categories and Subject Descriptors. *International Journal on Digital Libraries* (2015).
- [11] CHROMIUM. CookieMonster. <https://www.chromium.org/developers/design-documents/network-stack/cookiemonster>.
- [12] CZYZ, J., ALLMAN, M., ZHANG, J., IEKEL-JOHNSON, S., OSTERWEIL, E., AND BAILEY, M. Measuring IPv6 Adoption. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 87–98.
- [13] D. KRISTOL, L. M. HTTP State Management Mechanism, Oct. 2000. <https://tools.ietf.org/html/rfc2965.html>.
- [14] DHAWAN, M., KREIBICH, C., AND WEAVER, N. The Priv3 Firefox Extension. <http://priv3.icsi.berkeley.edu/>.
- [15] ECKERSLEY, P. How unique is your web browser? In *Proceedings of the International Conference on Privacy Enhancing Technologies* (2010).
- [16] ELECTRONIC FRONTIER FOUNDATION. Privacy Badger. <https://www.eff.org/privacybadger>.
- [17] ELTGROTH, D. R. Best Evidence and the Wayback Machine: a Workable Authentication Standard for Archived Internet Evidence. *78 Fordham L. Rev.* 181. (2009), 181–215.
- [18] ENGLEHARDT, S., EUBANK, C., ZIMMERMAN, P., REISMAN, D., AND NARAYANAN, A. OpenWPM: An automated platform for web privacy measurement. Tech. rep., Princeton University, Mar. 2015.
- [19] ENGLEHARDT, S., REISMAN, D., EUBANK, C., ZIMMERMAN, P., MAYER, J., NARAYANAN, A., AND FELTEN, E. W. Cookies That Give You Away: The Surveillance Implications of Web Tracking. In *Proceedings of the 24th International World Wide Web Conference* (2015).

- [20] EUBANK, C., MELARA, M., PEREZ-BOTERO, D., AND NARAYANAN, A. Shining the Floodlights on Mobile Web Tracking — A Privacy Survey. In *Proceedings of the IEEE Workshop on Web 2.0 Security and Privacy* (2013).
- [21] FOUNDATION, P. S. 21.24. http.cookiejar Cookie handling for HTTP clients, Feb. 2015. <https://docs.python.org/3.4/library/http.cookiejar.html>.
- [22] FREDRIKSON, M., AND LIVSHITS, B. RePriv: Re-Envisioning In-Browser Privacy. In *Proceedings of the IEEE Symposium on Security and Privacy* (2011).
- [23] GHOSTERY. Ghostery. <https://www.ghostery.com>.
- [24] GUHA, S., CHENG, B., AND FRANCIS, P. Challenges in measuring online advertising systems. In *Proceedings of the ACM Internet Measurement Conference* (2010).
- [25] GUHA, S., CHENG, B., AND FRANCIS, P. Privad: Practical Privacy in Online Advertising. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2011).
- [26] HACKETT, S., PARMANTO, B., AND ZENG, X. Accessibility of Internet Websites Through Time. In *Proceedings of the 6th International ACM SIGACCESS Conference on Computers and Accessibility* (New York, NY, USA, 2004), Assets '04, ACM, pp. 32–39.
- [27] HAN, S., JUNG, J., AND WETHERALL, D. A Study of Third-Party Tracking by Mobile Apps in the Wild. Tech. Rep. UW-CSE-12-03-01, University of Washington, Mar. 2012.
- [28] HAN, S., LIU, V., PU, Q., PETER, S., ANDERSON, T. E., KRISHNAMURTHY, A., AND WETHERALL, D. Expressive Privacy Control with Pseudonyms. In *SIGCOMM* (2013).
- [29] HANNAK, A., SAPIEŻYŃSKI, P., KAKHKI, A. M., KRISHNAMURTHY, B., LAZER, D., MISLOVE, A., AND WILSON, C. Measuring Personalization of Web Search. In *Proceedings of the International World Wide Web Conference* (2013).
- [30] IHM, S., AND PAI, V. Towards Understanding Modern Web Traffic. In *Proceedings of the ACM Internet Measurement Conference* (2011).
- [31] INTERNET ARCHIVE. Wayback Machine. <https://archive.org/>.
- [32] JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. C. Protecting Browser State From Web Privacy Attacks. In *Proceedings of the International World Wide Web Conference* (2006).
- [33] JANG, D., JHALA, R., LERNER, S., AND SHACHAM, H. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the ACM Conference on Computer and Communications Security* (2010).
- [34] JENSEN, C., SARKAR, C., JENSEN, C., AND POTTS, C. Tracking website data-collection and privacy practices with the iWatch web crawler. In *Proceedings of the Symposium on Usable Privacy and Security* (2007).
- [35] JOHN, N. A. Sharing and Web 2.0: The emergence of a keyword. *New Media & Society* (2012).
- [36] JONES, S. M., NELSON, M. L., SHANKAR, H., AND DE SOMPEL, H. V. Bringing Web Time Travel to MediaWiki: An Assessment of the Memento MediaWiki Extension. *CoRR abs/1406.3876* (2014).
- [37] KAMKAR, S. Evercookie—virtually irrevocable persistent cookies. <http://sam.y.pl/evercookie/>.
- [38] KELLY, M., BRUNELLE, J. F., WEIGLE, M. C., AND NELSON, M. L. On the Change in Archivability of Websites Over Time. *CoRR abs/1307.8067* (2013).
- [39] KONTAXIS, G., POLYCHRONAKIS, M., KEROMYTIS, A. D., AND MARKATOS, E. P. Privacy-preserving social plugins. In *USENIX Security Symposium* (2012).
- [40] KRISHNAMURTHY, B., NARYSHKIN, K., AND WILLS, C. Privacy Leakage vs. Protection Measures: The Growing Disconnect. In *Proceedings of the IEEE Workshop on Web 2.0 Security and Privacy* (2011).
- [41] KRISHNAMURTHY, B., AND WILLS, C. On the leakage of personally identifiable information via online social networks. In *Proceedings of the ACM Workshop on Online Social Networks* (2009).
- [42] KRISHNAMURTHY, B., AND WILLS, C. Privacy Diffusion on the Web: a Longitudinal Perspective. In *Proceedings of the International World Wide Web Conference* (2009).
- [43] KRISHNAMURTHY, B., AND WILLS, C. E. Generating a Privacy Footprint on the Internet. In *Proceedings of the ACM Internet Measurement Conference* (2006).
- [44] KRISTOL, D., AND MONTULLI, L. RFC 2109 - HTTP State Management Mechanism, 1997. <https://tools.ietf.org/html/rfc2109>.
- [45] LÉCUYER, M., DUOCOFFE, G., LAN, F., PAPANCEA, A., PETSIOS, T., SPAHN, R., CHINTREAU, A., AND GEAMBASU, R. XRay: Enhancing the Web's Transparency with Differential Correlation. In *23rd USENIX Security Symposium* (2014).
- [46] LEON, P. G., UR, B., WANG, Y., SLEEPER, M., BALEBAKO, R., SHAY, R., BAUER, L., CHRISTODORESCU, M., AND CRANOR, L. F. What Matters to Users? Factors that Affect Users' Willingness to Share Information with Online Advertisers. In *Symposium on Usable Privacy and Security* (2013).
- [47] LEONTIADIS, N., MOORE, T., AND CHRISTIN, N. A nearly four-year longitudinal study of search-engine poisoning. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 930–941.
- [48] LUND, A. The History of Online Ad Targeting, 2014. <http://www.sojern.com/blog/history-online-ad-targeting/>.
- [49] MAYER, J., AND NARAYANAN, A. Do Not Track. <http://donottrack.us/>.
- [50] MAYER, J. R., AND MITCHELL, J. C. Third-Party Web Tracking: Policy and Technology. In *Proceedings of the IEEE Symposium on Security and Privacy* (2012).
- [51] McDONALD, A. M., AND CRANOR, L. F. Americans' Attitudes about Internet Behavioral Advertising Practices. In *Proceedings of the Workshop on Privacy in the Electronic Society* (2010).
- [52] MILNE, G. R., AND CULNAN, M. J. Using the content of online privacy notices to inform public policy: A longitudinal analysis of the 1998-2001 US Web surveys. *The Information Society* 18, 5 (2002), 345–359.
- [53] MURPHY, J., HASHIM, N. H., AND OCONNOR, P. Take Me Back: Validating the Wayback Machine. *Journal of Computer-Mediated Communication* 13, 1 (2007), 60–75.
- [54] NARAIN, R. Windows XP SP2 Turns 'On' Pop-up Blocking, 2004. <http://www.internetnews.com/dev-news/article.php/3327991>.
- [55] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security* (2012).
- [56] NIKIFORAKIS, N., JOOSEN, W., AND LIVSHITS, B. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web* (2015), International World Wide Web Conferences Steering Committee, pp. 820–830.
- [57] NIKIFORAKIS, N., KAPRAVELOS, A., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy* (2013).
- [58] RESEARCH LIBRARY OF LOS ALAMOS NATIONAL LABORA-

- TORY. Time Travel. <http://timetravel.mementoweb.org/about/>.
- [59] REZNICHENKO, A., AND FRANCIS, P. Private-by-Design Advertising Meets the Real World. In *Proceedings of the ACM Conference on Computer and Communications Security* (2014).
- [60] ROESNER, F., KOHNO, T., AND WETHERALL, D. Detecting and Defending Against Third-Party Tracking on the Web. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2012).
- [61] ROESNER, F., ROVILLOS, C., KOHNO, T., AND WETHERALL, D. ShareMeNot: Balancing Privacy and Functionality of Third-Party Social Widgets. *USENIX ;login:* 37 (2012).
- [62] SOSKA, K., AND CHRISTIN, N. Automatically detecting vulnerable websites before they turn malicious. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 625–640.
- [63] STEVEN ENGLEHARDT. Do privacy studies help? A Retrospective look at Canvas Fingerprinting. <https://freedom-to-tinker.com/blog/englehardt/retrospective-look-at-canvas-fingerprinting/>.
- [64] TOUBIANA, V., NARAYANAN, A., BONEH, D., NISSENBAUM, H., AND BAROCAS, S. Adnostic: Privacy Preserving Targeted Advertising. In *Proceedings of the Network and Distributed System Security Symposium* (2010).
- [65] UR, B., LEON, P. G., CRANOR, L. F., SHAY, R., AND WANG, Y. Smart, useful, scary, creepy: perceptions of online behavioral advertising. In *8th Symposium on Usable Privacy and Security* (2012).
- [66] VISSERS, T., NIKIFORAKIS, N., BIELOVA, N., AND JOOSEN, W. Crying wolf? on the price discrimination of online airline tickets. In *HotPETS* (2014).
- [67] WAGNER, C., GEBREMICHAEL, M. D., TAYLOR, M. K., AND SOLTYS, M. J. Disappearing act: decay of uniform resource locators in health care management journals. *Journal of the Medical Library Association : JMLA* 97, 2 (2009), 122–130.
- [68] WANG, D. Y., SAVAGE, S., AND VOELKER, G. M. Juice: A Longitudinal Study of an SEO Botnet. In *NDSS* (2013).
- [69] WASHINGTON POST. From Lycos to Ask Jeeves to Facebook: Tracking the 20 most popular web sites every year since 1996. <https://www.washingtonpost.com/news/the-intersect/wp/2014/12/15/from-lycos-to-ask-jeeves-to-facebook-tracking-the-20-most-popular-web-sites-every-year-since-1996/>.
- [70] WILLS, C. E., AND TATAR, C. Understanding what they do with what they know. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society* (2012).
- [71] YEN, T.-F., XIE, Y., YU, F., YU, R. P., AND ABADI, M. Host Fingerprinting and Tracking on the Web: Privacy and Security Implications. In *Proceedings of the Network and Distributed System Security Symposium* (2012).
- [72] ZACK WHITTAKER. PGP co-founder: Ad companies are the biggest privacy problem today, not governments, 2016. www.zdnet.com/article/pgp-co-founder-the-biggest-privacy-issue-today-are-online-ads/.
- navigator.cookieEnabled
 - navigator.doNotTrack
 - navigator.language
 - navigator.languages
 - navigator.maxTouchPoints
 - navigator.mediaDevices
 - navigator.mimeTypes
 - navigator.platform
 - navigator.plugins
 - navigator.product
 - navigator.productSub
 - navigator.userAgent
 - navigator.vendor
 - navigator.vendorSub
 - screen.availHeight
 - screen.availLeft
 - screen.availTop
 - screen.availWidth
 - screen.colorDepth
 - screen.height
 - screen.orientation
 - screen.pixelDepth
 - screen.width
 - CanvasRenderingContext2D.getImageData
 - CanvasRenderingContext2D.fillText
 - CanvasRenderingContext2D.strokeText
 - WebGLRenderingContext.getImageData
 - WebGLRenderingContext.fillText
 - WebGLRenderingContext.strokeText
 - HTMLCanvasElement.toDataURL
 - window.TouchEvent
 - HTMLElement.offsetHeight
 - HTMLElement.offsetWidth
 - HTMLElement.getBoundingClientRect

B Ecosystem Complexity

Figure 12 (on the next page) visually depicts the connections between entities in the tracking ecosystem that we observe in our datasets for 1996, 2000, 2004, 2008, 2012, and 2016: domains as nodes, and referral relationships as edges. Note that the visual organization of these graphs (with nodes in multiple tiers) is not meaningful and simply an artifact of the graph visualization software. Over time, the complexity and interconnectedness of relationships between third-party domains on the top 450 websites has increased dramatically.

A Fingerprint-Related JavaScript APIs

As described in Section 3, TrackingExcavator hooks a number of JavaScript APIs that may be used in fingerprint-based tracking and drawn from prior work [3, 4, 15, 56, 57]. The complete list:

- navigator.appCodeName
- navigator.appName
- navigator.appVersion

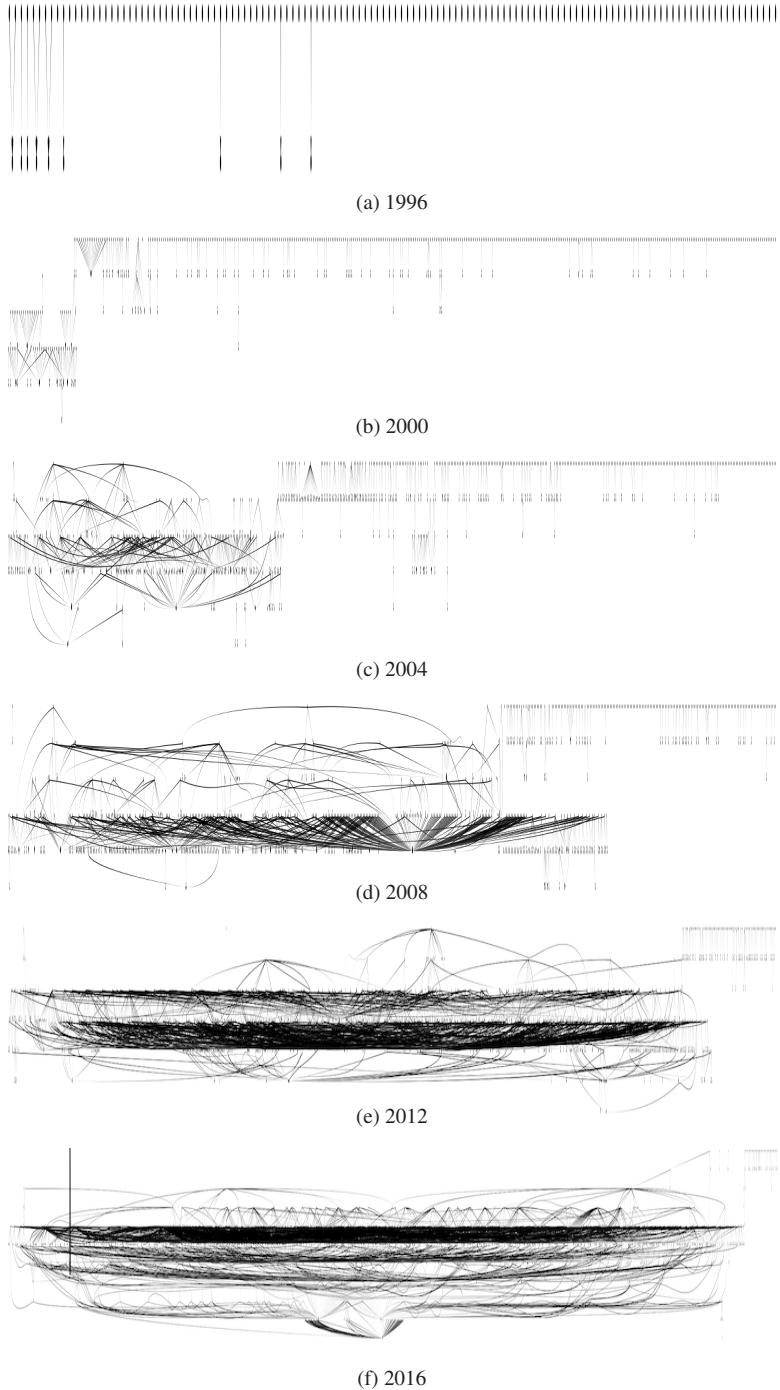


Figure 12: Referrer graphs for the top 450 sites in 1996, 2000, 2004, 2008, 2012 and 2016 as seen in the Wayback Machine’s archive. An edge from a domain `referrer.com` to another domain `referred.com` is included if any URL from `referrer.com` is seen to be the referrer for any request to `referred.com`. Note the increasing complexity of the graph over time.

Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification

Ben Stock

stock@cs.uni-saarland.de

*CISPA, Saarland University
Saarland Informatics Campus*

Giancarlo Pellegrino

gpellegrino@mmci.uni-saarland.de

*CISPA, Saarland University
Saarland Informatics Campus*

Christian Rossow

rossow@mmci.uni-saarland.de

*CISPA, Saarland University
Saarland Informatics Campus*

Martin Johns

martin.johns@sap.com

SAP SE

Michael Backes

backes@cs.uni-saarland.de

*CISPA, Saarland University & MPI-SWS
Saarland Informatics Campus*

Abstract

Large-scale discovery of thousands of vulnerable Web sites has become a frequent event, thanks to recent advances in security research and the rise in maturity of Internet-wide scanning tools. The issues related to disclosing the vulnerability information to the affected parties, however, have only been treated as a side note in prior research.

In this paper, we systematically examine the feasibility and efficacy of large-scale notification campaigns. For this, we comprehensively survey existing communication channels and evaluate their usability in an automated notification process. Using a data set of over 44,000 vulnerable Web sites, we measure success rates, both with respect to the total number of fixed vulnerabilities and to reaching responsible parties, with the following high-level results: Although our campaign had a statistically significant impact compared to a control group, the increase in the fix rate of notified domains is marginal.

If a notification report is read by the owner of the vulnerable application, the likelihood of a subsequent resolution of the issues is sufficiently high: about 40%. But, out of 35,832 transmitted vulnerability reports, only 2,064 (5.8%) were actually received successfully, resulting in an unsatisfactory overall fix rate, leaving 74.5% of Web applications exploitable after our month-long experiment. Thus, we conclude that currently no reliable notification channels exist, which significantly inhibits the success and impact of large-scale notification.

1 Introduction

The large-scale detection of vulnerabilities in Web applications has become significantly more common over the course of the last years. This can be attributed to two concurrent developments: The ever-growing adoption of open-source Web frameworks and the recent advances in automated vulnerability detection.

Open-source Web application frameworks, such as Joomla, Drupal, or WordPress, nowadays constitute the technological basis for a vast number of Web sites. Thus, a single vulnerability in any of these frameworks makes tens of thousands of Web applications attackable at once. Previously disclosed vulnerabilities range from Cross-Site Scripting attacks [20], to more severe attacks like SQL injections [10] or object deserialization flaws [19]. Given the rise in maturity and efficiency of Internet-wide scanning tools, such as ZMap [12], such flaws can effectively be identified on affected Web sites.

Furthermore, a recent stream of security research has demonstrated automated approaches that are capable of discovering Web vulnerabilities on a large scale. Examples of such research results include the discovery of high numbers of Client-Side Cross-Site Scripting problems [22], server-side application logic flaws [9], and instances of vulnerable server-side infrastructure [13].

However, as soon as a researcher has discovered security-critical vulnerabilities that affect thousands of Web sites, she has an ethical dilemma: On the one hand, with the awareness of a vulnerability comes the implicit responsibility of disclosing it to the affected parties. On the other hand, large-scale Web vulnerability disclosure is non-trivial, has not been well-studied, and there are no guidelines or suggestions on how to proceed.

Thus, in this paper, we explore the following questions: *Is it actually feasible to disseminate vulnerability information on a large scale? What are suitable communication channels to reach out to the affected parties? And does such a large-scale notification campaign affect the prevalence of vulnerabilities in the wild?*

To answer these questions, we conduct the first in-depth study on the feasibility and efficacy of large-scale Web vulnerability notification campaigns. To this end, we establish a large body of vulnerable Web sites, suffering from different types of vulnerabilities such as Reflected or Client-Side Cross-Site Scripting bugs. We then review communication channels to notify affected par-

ties at scale, including direct contacts (such as generic email aliases and WHOIS contacts) and indirect channels (such as hosting providers or CERTs). We notify the Web site administrators via these communication channels and compare how the vulnerabilities in the monitored Web sites evolve over time compared to a control group of vulnerable Web sites that we do not notify.

Our large-scale experiments reveal important take-away messages for fellow researchers. First, we show that while large-scale notifications can have a statistically significant impact on the fix rate of vulnerable Web applications, the long-term impact is marginal. Second, we therefore analyze the efficacy of direct and indirect communication channels, both with respect to reaching the (initial) recipient of the notification as well as the impact of the channel on the vulnerability landscape. Third, from our results, we derive the core challenges of notification campaigns and find that failures to reach out to contact persons cause the most severe degradation of success rates. Finally, we cover the lessons we learned during our pioneering efforts in order to assist researchers in future large-scale notification campaigns.

To sum up, we make the following contributions:

- We systematically evaluate the suitability of different communication channels for large-scale vulnerability disclosure and present a methodology to measure the feasibility and efficacy of such a disclosure process (§3).
- We document the results of a large-scale notification campaign, discussing the data set of vulnerable domains (§4) and the impact our campaign of notifying 35,832 vulnerable domains had (§5).
- Based on the observed (low) impact on the global landscape, we analyze factors inhibiting the success of large-scale disclosure campaigns (§6).
- We highlight key insights gathered in our study and present directions for future work in this space (§7).

2 Problem Statement

In this section, we outline the research questions we aim to answer. Secondly, we follow up with information on the actual vulnerabilities we consider for our work.

2.1 Research Questions

The ad-hoc process of reporting individual Web-related vulnerabilities to a single vendor is fairly well understood. Given its small scale, such a notification can be conducted with some manual, site-specific effort. However, detecting large numbers of vulnerabilities spanning several parties has become the norm rather than

the exception. Examples in the recent past include the identification of high numbers of vulnerable SSL implementations [13], Client-Side XSS vulnerabilities [22], or execute-after-redirect flaws in Ruby on Rails applications [9]. With higher numbers of affected parties, manual effort becomes unfeasible, and hence responsible disclosure transforms into a distinctly different problem. Therefore, it is necessary to investigate notification processes that function with little to no human involvement.

Taking this overarching motivation into consideration, we reach a set of distinct research questions: For one, given a large number of vulnerabilities that affect a similarly large set of disjoint site owners: *How can a scalable responsible vulnerability notification process be conducted?* As manual effort is not a realistic option, an automated vulnerability disclosure process is required.

Furthermore: *What are suitable communication channels to report vulnerabilities?* Not all methods to communicate vulnerability information can be easily used in automated processes. Once suitable channels are selected, it is necessary to examine *how successful a large-scale Web vulnerability disclosure process can be*. Even in an automated fashion, a large-scale notification campaign results in considerable effort: The notification process has to be set up, executed, and monitored. Moreover, even automated initial notification will in many cases lead to personal, non-automated communication with a subset of recipients of the vulnerability reports. Thus, it is reasonable to examine how significant the positive effect of our campaign was. Based on the results of such a study, another question arises: *Which methods were most successful in delivering notification, and what might inhibiting factors be?*

An increasing number of security researchers (professional and academic) spend significant time and effort to discover and fix security-critical software bugs, often wondering about how to proceed with large-scale disclosure processes. With the aforementioned research questions, we aim to shed light on the under-explored issue of *whether, how and with what chance of success* researchers can notify affected parties.

2.2 Vulnerability Information

To answer these research questions, we leverage a data set with concrete instances of Web sites that contain security-critical flaws. We can use this data set to notify affected parties and monitor their reactions. In the following, we outline the Web-related vulnerabilities we consider for our work, separated into *well-known* and *previously-unknown* vulnerabilities.

We establish the data set on well-known vulnerabilities by inspecting WordPress-based Web sites. We selected WordPress since it is the most frequently used

PHP-based web application, deployed by about 25% of the most popular Web sites [35]. To find WordPress vulnerabilities, we systematically searched the CVE (Common Vulnerabilities and Exposures) database [27] for vulnerabilities which could be verified with (i) non-intrusive proof-of-concept (*PoC*) tests and (ii) without requiring valid user credentials for the tested site. We chose two vulnerabilities: one reflected XSS from 2013 (CVE-2013-0237) and one Client-Side XSS discovered in 2015 (CVE-2015-3429).

In addition, we selected a recent vulnerability discovered by security researchers at Sucuri. This third vulnerability, which targets the XML-RPC service of WordPress prior to version 4.4, allows an attacker to perform brute-force amplification attacks [6]. The flaw is in the behavior of the XML-RPC service which accepts *multiple* remote procedure calls (RPCs) with *different* user credentials within a single HTTP request. As a result, an attacker can forge a request in which she tries several user passwords at once. In the remainder of this paper, we refer to this vulnerability as Multicall. All these vulnerabilities were already known and patched in current versions of WordPress when we started our experiments. In the following, we refer to these domains as Dwp.

Our data set on previously-unknown vulnerabilities contains Web sites from the Alexa Top 10,000 suffering from one or more Client-Side Cross-Site Scripting (*Client-Side XSS*) vulnerabilities. To detect these flaws, we used a methodology presented in our previous work (see Lekies et al. [22]) which is based on a taint-aware browsing engine and an exploit generator to gather verified exploitable vulnerabilities. To the best of our knowledge, these flaws were not shared with the site operators before our notification. The second data set thus represents a situation researchers face when discovering a previously unknown type or instance of Web flaws. We denote these domains as Dcxss.

3 Methodology

In this section we cover fundamental aspects of our methodology on large-scale notifications. We first discuss which communication channels can be used to reach out to affected parties. Subsequently, we outline how we prepared the notification messages. Third, we present the metrics that help to answer our research questions. Finally, we discuss ethical aspects of our methodology.

3.1 Communication Channels

The first key challenge when disclosing a Web vulnerability is to *reach out to an appropriate contact person*, e.g., to the administrator of a vulnerable Web site. In the

following, we review potential communication channels and discuss which of them are suitable in our context.

3.1.1 Direct Channels

We first consider and assess direct communication channels that lead to the responsible contact.

Web contacts (*discarded*) — One option would be to browse the Web site and search for contact email addresses, phone numbers, or contact forms. Naturally, calling affected parties or interacting with custom Web forms is not a viable option in a large-scale scenario. Alternatively, we could crawl the domain to extract email addresses. This process, however, is an unreliable and error-prone undertaking, and is complicated by anti-scraping mechanisms such as CAPTCHAs or obfuscated email addresses. Hence, such contacts are not viable for large-scale notifications and we do not consider them.

Generic email aliases — Standardized email aliases for each domain should ideally redirect to appropriate mailboxes and are “*to be used when contacting personnel of an organization*” (RFC 2142 [7]). The RFC also proposes alias categories suitable for our goal. Besides the security-related aliases, security@ and abuse@, we chose to contact the support mailbox for the HTTP service, i.e., webmaster@. In addition, we include the generic info@ alias.

Domain WHOIS information — The Domain WHOIS protocol can be used to query information about registered domain names. Depending on the providing server, however, the structure and content of the provided information varies. WHOIS data is optimized for readability to humans [8] and thus does not have a consistent document format [25]. While a human can easily use WHOIS to retrieve contact information for a *single* domain, it does not scale to large-scale disclosure. WHOIS providers also rate-limit requests and partially employ CAPTCHAs to protect contact addresses [17]. Hence, querying WHOIS at large scale is not feasible. Instead, to retrieve the WHOIS domain contact, we purchased a machine-readable WHOIS data set for the Alexa top one million Web sites as of September, 29th, 2015¹ and augmented the data with additional, on-demand queries. To select the contact person, first priority was given to the registrant’s email address. In cases where this did not exist, we selected the technical contact address instead.

3.1.2 Indirect Channels

Apart from direct channels, we can ask intermediaries to forward information about vulnerabilities on our behalf:

¹We bought the data set from <http://whoisxmlapi.com>

VRPs (*discarded*) — In recent years, vulnerability reward programs (VRPs) have gained traction and their success has been studied by researchers [14]. VRPs incentivize researchers to responsibly disclose flaws either directly to the vendor or to a VRP organization. Companies such as Google run their own in-house programs, whereas others outsource the coordination to organizations like HackerOne or BugCrowd. Naturally, in-house programs focus on vulnerabilities that are specific to the company, and hence, cannot be used for large-scale disclosure. Moreover, VRP organizations usually only accept and forward reports for their customers. Given a large body of vulnerable sites from several domains, VRPs are not a viable option for large-scale notifications. We thus exclude VRPs from our study.

Hosting providers — Hosting providers may already have an infrastructure in place to receive and react to security complaints. The provider likely has an incentive to use its direct customer contacts to forward the vulnerability information. While there is no specific security-related contact for providers, each provider typically has an abuse mailbox, which is used to notify operators about malicious activities originating from their networks. Abuse contacts can be found in Regional Internet Registries (RIRs) contact databases or queried via the IP WHOIS protocol. Both systems rate-limit requests and return proprietary formats. To work around this limitation, we query the Abusix Contact IP WHOIS proxy service [1] to obtain contacts of providers hosting the vulnerable Web sites.

TTPs — Trusted Third-Party Coordinators (hereafter TTPs) such as CERTs (Computer Emergency Response Teams) can act as intermediaries to report software vulnerabilities to operators. TTPs either operate on a regional level, i.e., within countries, or on a global scale, e.g., FIRST (Forum of Incident Response and Security Teams) [15]. Typically, TTPs already have technical infrastructure and procedures to forward vulnerability information to the administrators within their authority. To select CERTs, we determined the countries in which the vulnerable Web applications are hosted. We selected the top-20 countries in our data set and looked up their national CERTs. As global coordinators, we chose FIRST [15], and Ops-T [29] (Operations Security Trust), a closed community of security professionals.

3.2 Notification Procedure

In this section, we outline our notification procedure. Our campaign consisted of sending emails to the four notification groups on a bi-weekly basis, i.e., an initial notification and subsequent reminders every two weeks. In each round, we only notified Web sites which were

exploitable at least once in the previous 72 hours².

We split our overall data set of vulnerable Web sites into five disjoint groups of equal size, i.e., each domain is part of exactly one group. We use four notification groups and assign 1/5th of the domains to each of them: (i) generic email aliases, (ii) domain WHOIS contacts, (iii) abuse contacts of network providers, and (iv) TTPs. To reduce possible bias, we did not inform TTPs and network providers that they received only an excerpt of all affected sites in their constituency (since the rest were in the other groups). In addition, to set a baseline, we assign the fifth group to the *control group*, to which we did not send notifications.

3.2.1 Notification Types

To notify contacts, we carefully aggregated information to avoid a single contact receiving multiple notification emails, as discussed in the following.

Individual Disclosure — For the Generic and the WHOIS contact groups, we sent individual emails that contained a list of discovered flaws for their domain, as well as instructions to retrieve the technical report. We left it to the recipient to either view the technical report on our vulnerability disclosure Web interface, or to use our mailbot to retrieve the reports. Each domain therefore had a unique token assigned to it which could be used to retrieve the report. Additionally, the email informed the recipient that they could opt out of the experiments or get in touch with us via a dedicated mailbox. An example of the email is shown in Appendix A.

Aggregated Disclosure — For the network provider and TTP contact groups, the email contained a message similar to the one used for the individual disclosure, kindly asking the recipient to forward the information to the responsible domain admin. However, we aggregated vulnerability information per authority and prepared a single message to disclose multiple vulnerabilities affecting Web sites within the authority of TTPs and network providers. Specifically, we attached a CSV file specifying the domain, IP address, vulnerability types, Web interface link, and unique token for each site. The email also contained a note regarding opt-out and reaching us.

3.2.2 Mail Delivery and Anti-Spam Filters

To send out a large number of such notification emails, we opted to set up our own email server. Operating our own email infrastructure prevents side-effects that may have arisen when using existing email infrastructure of the university, e.g., in case an IP-based blacklist starts blocking our server due to the notification emails.

²See Section 3.3.1 for details on these exploitability checks

One of the key challenges for benign email campaigns is to avoid the emails being flagged as spam, based either on a bad sender reputation, or the message content. To minimize this risk, we implemented both Sender Policy Framework (SPF) [31] and, starting from the first reminder, DomainKeys Identified Mail (DKIM) [28]. For each email template, we used SpamAssassin to ensure that the message content would not be flagged as spam. Finally, throughout the mail sending process, we periodically monitored the reputation of our mail server by repeatedly querying IP-based blacklists, such as Spamhaus.org and SpamCop. In addition, as email providers may implement custom spam filters, we also tested our messages against the filters of the two most popular mail providers, i.e., Google Mail and Outlook, registering new email accounts on both services. In addition, we signed up for Microsoft’s Junk Mail Reporting Program (*JMRP*), which provides feedback on the spam check for mail from a given host [26].

3.2.3 Report Interface

During the experiment design time, one of the main concerns was the manual effort of our staff to address the quantity of possible questions that Web site owners might have. To help us with this type of activity, we built a web-based system, composed of a back end component for our staff and a front end for the Web site owner.

We wanted to provide users with as much detail as possible on the vulnerability, its impact, and ways of fixing it. Therefore, we created report templates for each vulnerability. For WordPress, we provided the details and hints on updating the installation. For Client-Side XSS, we provided the proof-of-concept URL which would open an alert box, as well as information on all the files which were involved in the exploitable data flow. Depending on the type of flaw, one of the customized templates would be presented to users of the front end.

The back end allowed us to retrieve current Web site statuses and statistics. Additionally, we automatically assigned emails to each domain. This allowed us to easily find all emails associated with a domain to give the best possible information on questions from domain admins.

3.3 Measurements

We broke down our research questions into a number of measurements that we perform throughout our notification campaign. These measurements are:

Global and Per-Group Vulnerable Web Sites — To measure the number of Web sites that are still exploitable, we set up a monitoring system which periodically verifies the exploitability of flaws using PoC tests.

In addition, as groups are mutually disjoint, our monitor naturally provides per-group results. The monitoring system is presented in Section 3.3.1.

Reachability of Recipients, Viewed Reports, and Time to Fix — We can directly measure the success of mail delivery by looking at both email responses and report interface access logs. The logs help to infer that a message has reached the recipient. We present this reachability analysis in Section 3.3.2.

3.3.1 Web Site Monitoring

Throughout our experiments, we periodically monitored Web sites to establish the point in time when they were no longer vulnerable. In the following, we discuss the different types and frequency of tests, as well as the approach used to determine that a site was fixed.

WordPress Vulnerabilities Tests — To test for the three WordPress vulnerabilities, we implemented the following vulnerability-specific probes.

XSS Vulnerabilities — The CVE-2013-0237 vulnerability affects a specific version of a Flash file which is part of the PIUpload component included in default WordPress installations. Our test retrieves the Flash file and compares its checksum against the checksum of the known vulnerable version. If the checksums match, the test returns *Exploitable*. In all other cases, it returns *Non-Exploitable*. Similarly, the presence of the CVE-2015-3429 vulnerability can be verified by comparing the checksum of a specific HTML page. If the checksum values match, the Web site is considered *Exploitable*; otherwise it is *Non-Exploitable*. Both these files have the same content regardless of Web site language, i.e., we did not have to implement a checker per site language.

Multicall — The detection of the Multicall vulnerability requires further care. In a vulnerable installation, the XML-RPC API checks all user-provided credentials per request. In the patched variant, it skips all credential checks if one has failed, but still returns a list of *invalid credential* error messages. As a result, the output of the service cannot be used to deduce exploitability. However, as vulnerable services process all calls—including the ones with invalid credentials—the processing time is longer than for the patched version. Based on this observation, we developed a test which uses this timing side channel to deduce if a site is vulnerable or not. For the technical details, we refer the interested reader to Appendix B. As side channels are susceptible to false positives, we correlate the results with the deployed version of WordPress, which we extract by using the testing tool *plecost* [18]. If both timing analysis and version reflect a vulnerable service, the site is *Exploitable*. In all other cases the Web site is *Non-Exploitable*.

Web Site Time-Series Analysis — Since the exploitability relies on core components of WordPress, it can be reliably triggered. To keep server loads to the necessary minimum, we only checked for these vulnerabilities once per day. Given the variety and number of Web sites that we monitor, however, our point-wise observations are susceptible to temporal errors, such as Web applications that are temporarily inaccessible or are otherwise unresponsive at the time of our check. To decide whether a Web site is no longer exploitable, or just temporarily unavailable, we calculate the confidence of our tests. The confidence is the complement of the number of unlikely events (i.e., number of observed transitions from *Non-Exploitable* to *Exploitable*). A site is only marked as fixed if the confidence is greater than 0.99. For a precise definition of our confidence function, we refer the interested reader to Appendix C.

Client-Side XSS Test — To test for this vulnerability, we used the set of per-domain exploits discovered with our methodology from previous work. Each exploit is a URL including the XSS payload. For more details on this aspect, we refer the reader to our paper [22]. The exploits are grouped according to the vulnerability they trigger. To keep the load on the target server low, we initially only check one exploit URL. If the exploit works, we consider the site still vulnerable. If the exploit fails, we do not consider the Web site as fixed yet. In fact, as our previous work has shown, a vulnerable page may no longer exist or the flaw may be caused by rotating advertisements [32]. To rule out such volatility, we re-check the exploit every three hours. Additionally, if the page no longer exists, we check other exploits from the same exploit group and update the PoC if any of them succeeds. A flaw is only marked as fixed if it was not exploitable for at least three consecutive days.

3.3.2 Mailbox and Report Access Log Analysis

One of our core research questions is to study the effectiveness of each notification group. To measure if we actually *reached* someone, we analyzed our mailbox and the logs of the front end. Our front end allows notified parties to retrieve a detailed technical report by clicking on a link, or, if the owner distrusts URLs in emails, via our mailbot. In both cases, the recipient has to submit a unique token. We kept track of all actions by logging the tokens and access times. This allowed us to perform fine-grained analyses regarding the access to our vulnerability reports. In addition, the mailbox we used to disseminate emails received a variety of automated replies that provided insights on the status of the email delivery. We use all this information to classify each contact point into one of the following categories.

Reached — For individual disclosure, we state that we *reached* a contact point when the response message is, for example, an auto-responder message acknowledging the receipt of our email, a response by a tracking system (e.g., a new ticket), and other messages in which the recipient unequivocally states that the message was received. We say that we reached a contact point also when we observe an access to corresponding domain’s technical report. In case of aggregated disclosure, if the email recipient, i.e., TTP or provider, is reached, then each of the domains within their constituency is also marked as reached. Similarly for individual disclosure, if a technical report for any domain within a constituency is accessed, we mark all associated domains as reached.

Bounced — Bounce messages are messages sent by a mail transfer agent to notify the sender that an error occurred and the message could not be delivered. Such errors might stem from the mailbox not existing or being full. If *all* emails we sent for a particular domain bounce, we classify the domain as *bounced*.

Unreachable — A contact point is unreachable when the response message indicates that no human will process our request. Examples of these cases are messages stating that the mailbox is unattended, or emails asking to contact Web site personnel only via a web form. Another example of an unreachable contact point is domains for which we could not retrieve any email address, e.g., if such information is missing in the WHOIS data.

Unknown — When we cannot establish whether a message was received, bounced, or the contact point was unreachable, we mark the contact point as *Unknown*.

While the identification of bounce messages is quite straightforward based on their content (e.g., SMTP error codes), automatically assigning emails to the aforementioned categories is prone to errors. Hence, we manually assigned incoming emails to one of the categories.

3.4 Ethical Considerations

We addressed ethical concerns from the early stages of our methodology design. In general, we design our experiments to be *unobtrusive*. This is, e.g., reflected on both the WordPress vulnerabilities selection criteria and the monitoring frequency. Despite that, our regular checks may still be undesired by network providers and Web site owners. For this reason, in our emails, we included instructions to *opt out* of our study. Moreover, we configured descriptive reverse DNS names for our infrastructure and hosted a website that described our initiative, again detailing contact information (postal address, email address, phone numbers) and an opt-out procedure.

The second ethical consideration of our experiments was fairness towards Web site administrators. As a result

of our methodology, there are administrators that were not made aware of vulnerabilities affecting their sites, i.e., were contained in the control group. After the end of our notification campaign, we informed them using the discussed direct channels and shared the list of vulnerable domains with the TTPs.

Our experiments are in part related to human operators on the receiving end of our notification emails. Our organizations, however, neither mandate nor provide an IRB approval before conducting such experiments.

4 Data Set

In this section, we explain the details of our experiments such as the time period, data sets of vulnerable applications, and composition of our notification groups.

Vulnerable Domains — In total, our data set included 44,790 Web sites of which 43,865 are WordPress-based Web sites suffering from at least one vulnerability discussed in Section 2.2 (D_{WP}). The remaining 925 Web sites were susceptible to site-specific Client-Side XSS exploits (D_{CXSS}).

Notification Groups — We randomly split the data sets of vulnerable Web sites into five equally-sized groups of 185 D_{CXSS} and 8,773 D_{WP} domains. During the lookup process, we could not retrieve contact points for several domains. For the domain group, the WHOIS database did not contain email addresses for 34 (18.4%) and 1,665 Web sites (19.0%) for D_{CXSS} and D_{WP}, respectively. For the network provider group, queries to the Abusix servers did not return a contact for 18 (9.7%) and 254 domains (2.9%) of D_{CXSS} and D_{WP}, respectively. In all these cases, we marked these Web sites as *unreachable* within their contact group.

For the TTP group, we followed a different approach. To select regional coordinators, we extracted the country code of the ASN hosting each domain, using the IP-to-ASN database of Team Cymru [33]. Finally, we selected coordinators for the 20 most frequent country codes from the list maintained by CERT-CC [4]. These regional CERTs account for 90.8% of the domains in the TTP contact group, the remaining 9.2% were hosted in a country outside of the top 20. To close this gap, we augmented the set of coordinators with the two global coordinators FIRST and Ops-T.

Notification Campaign — We notified the affected parties on Jan 14th, 2016. We sent two reminders, one after two weeks (Jan 28th), and one after four weeks (Feb 11th). For FIRST, we accidentally delayed the initial mail delivery by two days due to a misunderstanding, but made sure they received the vulnerability information as soon as the issue was resolved. In total, our server delivered 17,819 emails as the initial notification, 15,110 as

primary reminders, and 13,588 as secondary reminders. In each round, the number of emails sent decreased because administrators fixed the vulnerability, they explicitly asked to be excluded from the experiments, or email addresses were invalidated due to bounces.

Unsubscribed Domains — As discussed in Section 3.4, we enabled domains to opt out of our analysis. Throughout the duration of our experiments, we received five requests to exclude a total of 187 domains, of which 149 were in a notification group and 38 in the control group. We received an email from a hosting provider on the second day of our campaign, threatening to sue our university if we did not immediately stop the analysis on this network. Additionally, we received messages from domain owners that were contacted by their hosting providers, stating that their Web sites would be taken down if the vulnerabilities were not fixed within a short timeframe. In these cases, we not only excluded the domains from any further analysis, but also reached out to the providers to clarify the obvious misunderstanding.

5 Site Vulnerability Evolution

Using the methodology described before, we now instantiate our large-scale notification experiment and assess whether we can affect the prevalence of vulnerabilities in the wild. We answer this question by looking at the observed trends of fixed vulnerabilities and the significance of our campaign. Finally, we have a closer look at each data set and discuss the impact in isolation.

5.1 Trend of Fixed Vulnerabilities

Table 1 shows the total number of non-exploitable domains at the end of our measurements (February 16th). The fraction of non-exploitable Web sites ranges from about 25.1% (*Generic*) to 26.5% (*WHOIS*) for D_{WP}. For D_{CXSS}, the fraction of domains has a greater variety and ranges from 8.6% (*Provider*) to 16.8% (*TTP*). We observe a distinctive difference in the fix rates for the control group for D_{WP} (23.3%) and D_{CXSS} (2.4%). We discuss the reasons for this in Sections 5.2.1 and 5.2.2.

	D _{WP}	D _{CXSS}
Generic	2,201	25.1%
WHOIS	2,325	26.5%
Provider	2,261	25.8%
TTP	2,268	25.9%
Control	2,043	23.3%
	25	13.5%
	21	11.4%
	16	8.6%
	31	16.8%
	4	2.2%

Table 1: Non-exploitable domains per group by 02/16

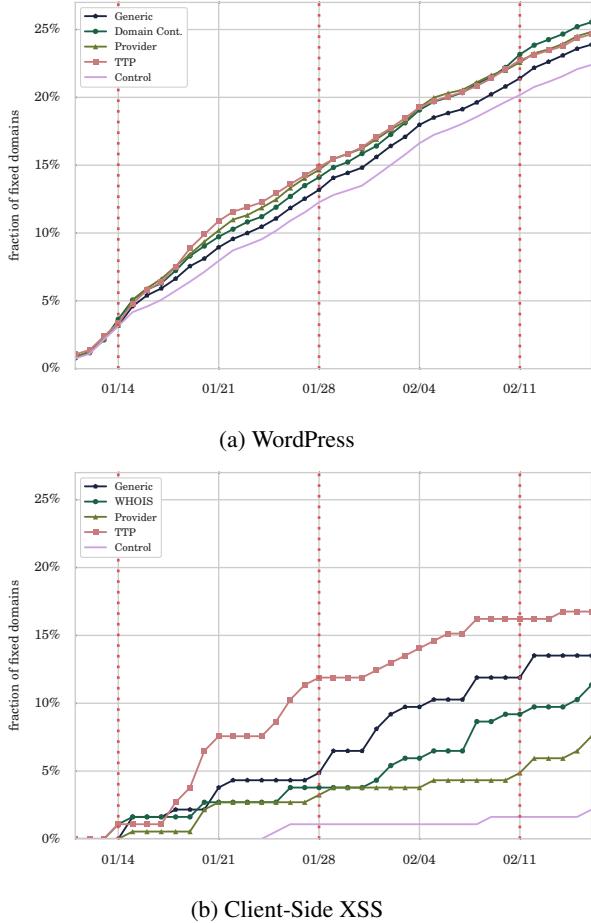


Figure 1: Fixed vulnerabilities over time

Figures 1a and 1b show the timeline and reveal when the vulnerabilities were fixed. For D_{WP}, we observe that all groups (including the control group) have a steady increase in non-exploitable domains. Each group progresses to the maximum, following a slight “wave” shape. This shape correlates to a weekly pattern. The weeks following a notification round (denoted as the horizontal lines) are characterized by a slightly steeper increase than the weeks before a notification round. For each of the groups, more than 2,000 of the 8,773 domains were no longer vulnerable at the end of our study. Overall, all notification groups performed better than the control group.

In contrast to the steady increase we observed for D_{WP}, no such pattern can be found for D_{CXSS}. Generally speaking, the difference between control and notified groups is much larger for D_{CXSS}. We discuss the results in Section 5.2.

Result Significance — During our experiments, we witnessed an increase in the fix rate for notified domains in comparison to the control group for both D_{WP} and

	D _{WP}	D _{CXSS}	
Generic	0.0053449	7.76	0.0000486
WHOIS	0.0000009	24.24	0.0004299
Provider	0.0001308	14.63	0.0058000
TTP	0.0000796	15.57	0.0000016
Overall Campaign	0.0000012	23.51	0.0000360
			17.07

Table 2: *p*-values and raw values from χ^2 tests

D_{CXSS}. In order to ascertain whether this is due to chance or an effect of our campaign, we analyze the gathered data in more detail. To that end, we state the hypothesis H_0 that the difference in the results originates purely from chance and is not an effect of our notification. For both D_{CXSS} and D_{WP}, we use Pearson’s χ^2 [30] test to calculate the *p*-values for each notification group in comparison to the control group. This test allows us to determine whether the differences between the results arose by chance. Both the resulting *p*-values and the raw χ^2 values for both D_{WP} and D_{CXSS} are shown in Table 2.

In our case, however, we need to account for the comparison of the control group to several other groups. Hence, an error in the control group would bias all comparisons. Applying the Holm-Bonferroni method ($\alpha = 0.05$) [16], we observe that every value p_i is below $\frac{\alpha}{5-i+1}$, i.e., H_0 does not hold. Hence, we find that the notified groups all differ significantly from the control group.

For the sake of completeness, in addition to the comparison to the control group, Tables 7 and 8 in the appendix show the *p*-values when comparing the notified groups against each other. While direct comparison between two groups sometimes reveals significant differences (e.g., D_{WP} WHOIS and Generic), no group performed significantly better than *all* other groups.

5.2 Impact Analysis

In this section, we analyze the collected data in more depth. More precisely, we discuss the changes for D_{WP} over time, showing how long-lasting the effect of our notification was. Moreover, we explain why the number of fixed domains in the control group is high, at almost one fifth. Subsequently, we discuss the impact of our notification on D_{CXSS}.

5.2.1 WordPress

Our notification campaign increased the number of domains that fixed the WordPress vulnerabilities by 11.2%. All notification groups outperformed the control group. The WHOIS group was the communication channel with the highest fix ratio (+15.4%). Next, the TTPs (+12.1%)

and providers (+10.9%) were second and third, respectively. The least effective channel was Generic, which still showed a 5.9% increase over the control group.

Besides the overall impact, we analyzed our collected data on a per-week basis, starting on the first day of our campaign. The results of this analysis are shown in Table 3. Next to the absolute number, the table also shows the fraction of domains fixed in each week. Note that this is relative to the number of domains that was still vulnerable at the beginning of that week for that group. In the first week, all channels performed better than the control group. In the second week, only providers still showed an observable increase in fixed domains. After the reminders, only the Generic and WHOIS contact groups had slightly increased fix numbers. All in all, however, the most drastic change occurred in the first week. Based on our definition of *fixed* domains (see Section 3.3.1), conclusive results can only be given at least three days back. As we stopped our experiments on January, 17th, we can only provide information on the first four weeks.

We observe that a substantial fraction of about one fifth of the control group was fixed during the duration of our study. This observation can be explained by analyzing the evolution of WordPress installations. We require version information to make a decision about the Multicall flaw, i.e., we had this information readily available for all days of the experiment. Indeed, out of the 8,773 domains in the control group, 1,134 moved from a version prior to 4.4 to an updated variant (not susceptible to Multicall) in the timeframe of our experiments. In addition, we observed that 360 domains no longer used WordPress or were offline at the end of our study. Throughout the remainder of the paper, we refer to these fixes as the *natural decay* of vulnerable domains.

It is also clear from Table 3 that in the first week of our campaign, a comparatively higher number of domains in the control group was marked first. While we cannot conclusively say why this occurred, we have anecdotal evidence from emails we received from one administrator who was responsible for multiple domains. Even though we had only notified him about one do-

main, he fixed several domains at once. Similarly, the WHOIS data set we purchased shows that different domains (spread across notified and control groups) contained the same contact email, i.e., had the same owner.

5.2.2 Client-Side XSS

Contrary to what we observed for DWP, there is no evidence for a natural decay of Client-Side XSS vulnerabilities. This stems from the fact that updates for the WordPress flaws are readily available and that sites are constantly being upgraded. For DCXSS, however, to the best of our knowledge, developers were not aware of the flaws and no automated update existed to patch the flaws. Therefore, the impact of our campaign is much higher in comparison to DWP.

In total, our campaign on average increased the number of fixed domains by a factor of almost 6, compared to the negligible number of *three* fixed domains in the control group. The highest fix rate was achieved by Trusted Third-Parties (16.8%), followed by the Generic channel with 13.5%. Additionally, 11.4% of the WHOIS and 8.7% of the provider group domains were fixed.

Important to note in this instance is one specific feature of Client-Side XSS: such issues are often caused by third-party scripts [32], which are out of the control of the administrator of the vulnerable domain. If a vulnerable third-party component is used across multiple domains, it is sufficient if *one* affected party reports this to the third-party vendor. In one particular case, we found that nine domains suffered from the same flaw. The vulnerability was fixed on February 8th, and its effect is visible in Figure 1b in the increase of fixed domains for Generic (three domains), WHOIS (four domains), and TTP (two domains) on that day. Since by chance, none of the domains was in the control group, this anomaly had a heavy impact on the overall notification campaign.

6 Communication Channel Analysis

In the previous section, we outlined the global picture on the vulnerability landscape and how much our campaign impacted it. Although the notifications for both DWP and DCXSS showed significant improvements over the control group, the number of domains which were fixed is unsatisfactory (25.8% and 12.6%, respectively). This raises the question whether we succeeded in reaching out to administrators. Therefore, in this section, we analyze how both direct and indirect communication channels performed in terms of successfully reporting the flaws to the responsible administrators.

	Control	Generic	WHOIS	Prov.	TTP
14/1-20/1	438 (5.1%)	521 (6.1%)	610 (7.2%)	631 (7.4%)	664 (7.8%)
21/1-27/1	387 (4.8%)	387 (4.8%)	397 (5.1%)	416 (5.3%)	395 (5.0%)
28/1-03/2	382 (5.0%)	416 (5.5%)	418 (5.6%)	380 (5.1%)	380 (5.1%)
04/2-10/2	386 (5.3%)	389 (5.4%)	402 (5.7%)	368 (5.2%)	365 (5.2%)

Table 3: WordPress flaws fixed per week

6.1 Direct Channels

In this section, we analyze the direct channels, first determining whether we reached the intended recipient. We then assess how many reports were accessed and how quickly flaws were fixed after report view.

6.1.1 Reachability Analysis

For the direct channels, i.e., Generic email addresses and WHOIS contacts, we sent a single email per domain. As outlined in Section 3.3.2, we then classified each domain as to its *reachability state*. The results of the classification are shown in Table 4. Based on the numbers we observe, several characteristics for the direct channels become apparent. First and foremost, more than half of all domains are marked as *unknown*, either because the emails were silently bounced, delivered to an unmonitored mailbox, or ignored by the recipient. Apart from this, we observe that the groups have distinct differences, which we discuss separately in the following.

For the Generic group, we received a large number of email bounces. More precisely, for 50% and 28% of DWP and DCXSS domains, respectively, *all* emails we sent bounced. The difference in number of bounces between DCXSS and DWP likely originates from the higher popularity of DCXSS Web sites (Alexa Top 10,000) in comparison to DWP (Alexa Top 1 million). As popular Web sites may have a more structured staff, they may tend to adhere to standards like RFC-2142 [7], thus reducing the number of bounces. Nevertheless, even for the high-ranked DCXSS Web sites, only 41 (22.2%) were actually reached. Moreover, for 90 DCXSS domains (48.6%) we neither received emails acknowledging our reports, nor saw any hits on our Web site. Similarly, we observe that more than 45% of the DWP domains are *unknown*.

The situation for the WHOIS group is slightly different: the fraction of bounces is significantly lower than for the Generic channels. This appears natural based on the fact that a valid email address is typically necessary to register a domain. However, apart from the large body of unknown domains, we see that the second-biggest fraction of domains belong to the *unreachable* bucket. This

is caused by the fact that for 1,699 domains (both DWP and DCXSS), we could not retrieve a contact address from the WHOIS information. Additionally, for 37 domains, our messages were not delivered to the domain owner because the emails from the WHOIS database are of organizations that hide email addresses. The remaining 31 domains in that bucket were marked as unreachable since we received emails redirecting us to a Web-based form.

Given the large volume of emails we sent out for direct notifications (four on Generic, one on WHOIS), anti-spam filters also interfered with reaching out to Web site administrators. Despite our careful preparation of infrastructure and email content, our messages were partially labeled as spam. Even though our mailserver was never listed in any well-known blacklist, Microsoft’s JMRP reported that the emails of the first two rounds were in parts flagged as spam. Interestingly, none of the emails of the second round of reminders were labeled as spam.

Additionally, provider-specific anti-spam filters mislabeled our emails. For 562 domains, we received bounces stating that our mails were classified as spam and thus rejected. In addition, in a handful of cases, we received feedback from contact points stating that they had only received our reminders and not the initial notification. This was either caused by *silent bounces* (the mail server accepted the email, but dropped it without notifying the sender), or by our email ending up in the spam folder, and then being automatically deleted after a few days.

6.1.2 Report Access

As we have seen in the previous section, the number of reached domains for the direct channels is low, amounting to only 357 and 714 DWP domains for Generic and WHOIS, respectively. The increase in fixed DWP domains compared to the control group, however, is even smaller: 158 and 282, respectively (see Table 1). We observe a similar trend for DCXSS. To investigate this discrepancy, in the following we analyze for how many of the *reached* domains a report was accessed on our Web interface or via the mailbot.

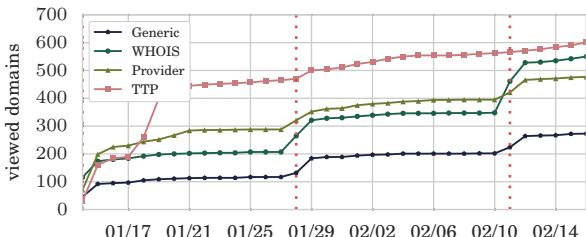
Table 5 shows the number of domains which accessed a report at least once. In addition, Figures 2a and 2b

	Generic		WHOIS		
	DWP	DCXSS	DWP	DCXSS	Total
Reached	357	41	714	38	1,150
Bounced	4,395	52	771	14	5,232
Unreach.	10	2	1,731	36	1,779
Unknown	4,011	90	5,557	97	9,755
Total	8,773	185	8,773	185	17,916

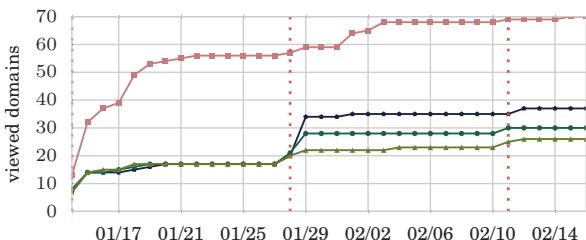
Table 4: Success of direct channels

	DWP	DCXSS
Generic	273	3.1%
WHOIS	550	6.3%
sum direct	823	4.6%
Provider	477	5.4%
TTP	601	6.9%
sum indirect	1,078	6.1%
	37	20%
	30	16.2%
	67	18.1%
	26	14.1%
	70	37.8%
	96	25.9%

Table 5: Viewed reports for all channels up to 02/16



(a) WordPress



(b) Client-Side XSS

Figure 2: Accessed reports for different channels

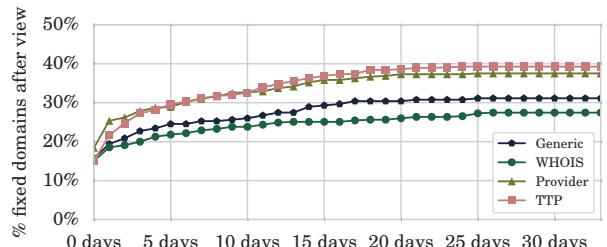
plot the temporal evolution of the viewed reports for DWP and DCXSS, respectively. We observe that within a few days of the initial notification, the number of accessed domain reports stabilizes for both DWP and DCXSS. Both reminders increased the number of viewed domains, but interestingly the effect for DWP was larger for the second reminder, whereas for DCXSS this held true for the first.

In total, only 823 of the DWP domains had reports viewed for the direct channels. For DCXSS, the ratio of viewed reports was much higher, but still adds up to only 67 (18.1%) of the notified domains, whereas Generic was slightly better than WHOIS with 37 viewed reports.

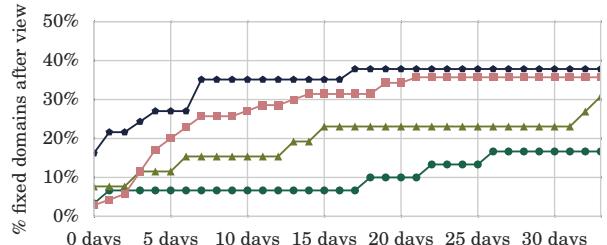
6.1.3 From Report Access to Fix

Even though not all *reached* domains had a report view, the number of domains for which a report was viewed is still larger than the increase in fixed domains. This is due to the fact that instead of only viewing a report, the final step towards ensuring that the domain is no longer vulnerable is to understand the specific issue and patch it accordingly. In this section, we therefore analyze how many domains from the direct channels were fixed after a report was viewed and how long it took.

Figures 3a and 3b show the time between a report being viewed and the fix of the underlying flaw. For DWP we observe on the Generic channel about 30% of the *viewed* domains are fixed within 5 days. Similarly, for WHOIS we observe a slightly higher rate of 32.5% within that timeframe. After this, the increase in fixed



(a) WordPress



(b) Client-Side XSS

Figure 3: Time from first report view to flaws fixed

domains aligns with what we observed for domains in the control group (roughly 0.5-0.7% per day). In essence, domains are either fixed within a very short time after initial report view or become non-exploitable just based on the natural decay of vulnerable domains.

For the DCXSS domains, this pattern differs, showing significant increases in the fix rates even after more than a week. These flaws, in contrast to WordPress, are mostly site-specific and, more importantly, a fix has to be developed first. Hence, a longer timeframe for the fix is somewhat expected. Of all channels, the Generic channel had the highest rate of fixed domains after having viewed a report with about 38%. Noteworthy in this instance is the poor conversion rate for the WHOIS group. One possible explanation for this is the fact that the domain owner for high-ranked sites might be disconnected from the engineering team. Hence, the information might have been viewed by this person, but not properly forwarded to the actual administrator of the site. In contrast, for DWP, we argue that such installations are mostly used by either small companies or single persons, and hence no such disconnect exists.

6.2 Indirect Communication Channels

After the discussion of direct channels, we now follow up with an in-depth analysis on the indirect channels.

	Provider		TTP		total
	DWP	Dcxss	DWP	Dcxss	
Reached	3,992	75	7,567	138	11,772
Bounced	2	0	0	0	2
Unreach.	271	20	0	0	291
Unknown	4,508	90	1,206	47	5,851
Total	8,773	185	8,773	185	17,916

Table 6: Success of indirect channels

6.2.1 Reachability Analysis

Similarly to direct channels, we also classified all domains based on their reachability. In this case, however, reaching the end point does not entail that we reached the administrator of the site. Instead, since we rely on intermediaries, we can only measure whether they received the email and not whether they forwarded it to the responsible party. This is also reflected in Table 6 in the large number of *reached* domains. As discussed in Section 3.3.2, a domain is marked as *reached* once the intermediary was reached, i.e., either confirmed our email or viewed at least one report disclosed to them.

In total, we reached 592 network providers responsible for 3,992 and 75 domains, respectively. Unreachable domains for the indirect channels were providers for which no contact existed in the Abusix database. In contrast to the high number of reached providers, about 50% for both DWP and Dcxss remain unknown.

For TTPs, such as CERTs or Ops-T, the numbers seem even more promising. Here, the *unknown* domains correspond to such domains for which we received no feedback from the hosting country’s CERT, or those which were not located in any of the top 20 countries. Since a relatively small number of TTPs is responsible for a large body of vulnerable domains, the fraction of *reached* domains is comparatively high.

6.2.2 Report Access

The large number of reached domains appears to be a positive sign for a successful vulnerability notification. However, looking at the number of accessed reports shown in Table 5, we find that the improvement over the direct channels is less significant.

In general, the report access pattern for the provider group (depicted in Figures 2a and 2b) is similar to what we observed for direct channels: an initial increment of access to our reports after each notification round followed by long intervals of a quasi-constant number of accesses. For the provider group, the percentage of viewed reports remains low at 5.4% and 14.1% for DWP and Dcxss, respectively. For the most part, this is caused by unhelpful providers: the top 5 providers accounted for

2,082 domains, but none of them reviewed any report, thus effectively stopping the notification process dead in its tracks for more than half of the *reached* domains.

Compared to the providers, TTPs show a significantly different access pattern. First of all, after the first notification round, we observe an increase of access after four days. This can in part be attributed to FIRST receiving our initial email two days late. Additionally, from feedback received from the third-largest active regional CERT, we learned that they follow their own dates to distribute notifications. While we sent notifications every other Thursday, this CERT sent their notifications on Mondays. In addition, we argue that the CERTs did not simply forward our messages, but rather vetted them first. Therefore, it is highly likely that the information was vetted on Friday, and only forwarded to responsible parties on Monday. This can be observed in Figures 2a and 2b as the steep increase starting from the fourth day of our campaign (i.e., Monday, January 18th).

For TTPs, we have a large number of reached domains, but cannot observe an analogous increase in number of viewed reports. The fractions of viewed reports are 37.8% for Dcxss and 6.9% for DWP. In contrast to the direct channels, our measurements could not reveal direct causes for these low numbers such as bounces or unreachable contacts. An explanation may be that TTPs did not forward our notification emails to Web site administrators. Among the 20 regional CERTs, 18 reacted to our email. Since we did not receive bounces for the two non-reactive CERTs, we marked the domains in their constituency as *unknown*. 10 CERTs that reacted to our email did not view a single report. This could happen for two reasons. First, rather than vetting the information, these TTPs could have directly forwarded the information, but the Web site administrators did not receive them, or did not act upon them. Second, the CERT might not have forwarded the information at all. Given the assumption that a TTP would first vet the information originating from an untrusted source, the lack of accesses to the report favors the second explanation, i.e., our notification messages were not forwarded by half of the CERTs.

In total, the combination of indirect channels performed better in terms of accessed reports than the combination of direct channels. However, providers performed worst of all channels for Dcxss and ranked third for DWP. In contrast, TTPs were most successful for report access on both types of vulnerable domains. This result, however, is greatly influenced by Ops-T: this TTP alone was exclusively responsible for 135 report accesses for DWP and 36 for Dcxss. Let us consider a scenario in which we could have relied only on regional CERTs and FIRST. In this case, the number of viewed reports for TTPs would have gone down to 466 and 34 for DWP and Dcxss, respectively. These numbers are similar to

or even lower than the other groups. Hence, although the TTPs were of great help in our campaign, a researcher without access to the vetted Ops-T community could not have achieved a comparable report access rate.

6.2.3 From Report Access to Fix

For the indirect channels, we also measured the time between first access of a report and fix for the disclosed vulnerability. The results are depicted in Figures 3a and 3b. For DWP, although TTPs perform similar to the direct channels, they have a distinct lag. More precisely, only 15.4% of domains were fixed within one day of report access, whereas this number ranged between 20.8% and 22.4% for direct channels. This underlines our hypothesis that TTPs would first vet the information we presented them. Hence, the first access to a domain report would have originated from the TTP, which subsequently forwarded the information to the responsible party.

As discussed before, there is less of a natural decay for DCXSS vulnerabilities, hence any fix is more likely to be caused by our notifications. However, due to the specifics of Client-Side XSS, where a single third-party script may be the cause for multiple flaws, it is hard to detect a distinct trend for the vulnerabilities. Similar to the DWP flaws, we observe a lag for both providers and TTPs between the initial view of the report and the time to fix, especially compared to the Generic group. This again highlights the vetting process of the intermediaries.

6.3 Discussion

As this section highlighted, the most significant issue we were faced with during our notification campaign was reaching the administrators in the first place. The issues depend on both communication channels and characteristics for the vulnerable domains. Naturally, reaching the intermediaries was quite straightforward: the email addresses were well-known and we only needed to send a comparatively small number of emails to them. In contrast, on the direct communication channels, especially for domains from DWP, we had a large number of bounces or unreachable contacts to begin with.

While the results for reachability suggest that using intermediaries greatly improves the chances of successful notifications, the benefit was not carried on to the number of viewed reports. Although TTPs performed best for both DWP and DCXSS report views, this was mostly caused by the closed Ops-T community. Additionally, our reminders improved on the number of accessed reports, especially for the direct channels.

Once a report was viewed, fix rates for DWP were similar across all groups, while the providers had a slightly higher fix rate. Moreover, we found that with a chance

of approximately 30%, domains were fixed within 5 days of a report view. After this period, the fraction of fixed domains only increased by what we observed to be the natural decay of flaws. For DCXSS, differences between communication channels were more drastic, with a much lower performance by the WHOIS channel. Although the number of fixes is subject to side-effects from vulnerabilities caused by third-party scripts, we note that the Generic channel worked best when considering the fix rate within the first 5 days.

7 Key Insights and Follow-Up Questions

In our work, we not only wanted to measure how successful a large-scale vulnerability notification campaign could be, but also aimed at determining what issues researchers would face. In the following, we discuss the key insights gained in our efforts, and present a number of follow-up questions which arise out of our findings.

Establishing Communication Channels — First and foremost, establishing a direct communication channel is remarkably challenging. For direct channels, we observe three main problems: (i) standardized addresses perform poorly with less popular Web sites; (ii) WHOIS contacts are a valid alternative for less popular Web sites, but the WHOIS database is not complete (about 20% domains lack contact points); and (iii) sending a large number of emails can be considered a spam campaign. All these reasons contribute to the low fraction of viewed reports.

Relying on indirect channels may reduce the workload for disclosure, as it *de facto* outsources the effort to an external organization. We found that for a large fraction of the domains such an intermediary could be reached. Although judging from the number of viewed reports, TTPs have the highest success rate, the overall results are still unsatisfactory. Moreover, we cannot ascertain how many TTPs forwarded the information or simply discarded them. We also found that reminders do not cause significant changes for TTPs, but have a slight impact on providers. Given all these facts, we find that establishing a communication channel to Web site administrators remains a hard problem even in the presence of intermediaries. Thus, the first question that arises for future work is: *How can the security community come up with reliable means of establishing communication channels between researchers and affected parties?*

User Distrust — For our experiments it was imperative that we were able to keep track of delivered and viewed reports. Therefore, we embedded a link to our web interface into the email. This naturally increases the risk of improper spam classification. Moreover, the security community trains users not to click on untrusted links or respond to suspicious emails, i.e., a significant fraction

of all reached administrators might not have followed our link or accessed the report via email. To investigate to what extent such behavior might have influenced our findings, we sent emails containing the full vulnerability details to all domains of the control group that were still vulnerable at the end of our experiments, using only the direct notification channels. While for Dcxss, no significant difference could be observed, Dwp domains notified this way show significant differences. Contrary to the intuition that a report only accessible via a link would hinder the campaign, however, these domains performed *worse* in terms of fix rates. This curious fact might be caused by the fact that the emails contained multiple links (e.g., to the vulnerable site, the description on mitre.org, and the information on updating WordPress), hence triggering more spam filters. Alternatively, such long emails might have aroused more suspicions by the recipients. Hence, this rises another question: *To what extent does the message tone, content, and length influence the success of notification campaigns?*

Sender Reputation — When looking at the results for TTPs in more detail, we find that the trusted Ops-T community was responsible for a large portion of successful notification deliveries, even up to 50% of Dcxss viewed reports for the TTPs. Thus, we argue that although studies have shown otherwise [5], trust in the sender of a notification message may be important factor to a campaign’s success. This also holds true for the German CERT, which (according to our data) was more inclined to forward our messages. In this case, the cause is most likely the fact that we originate from a German university and have had interactions with the CERT before. This brings up another question for future work: *What is the impact of the sender reputation, especially when using intermediaries, on the success of a notification campaign?*

Time to Fix and Need for Reminders — Once a report was viewed, the fix rate for Dwp was around 30% within five days, regardless of the channel that was used to originally transmit the report. After that, the fix rate approximates what we observe for the control group, i.e., is most likely not an effect of our notification. For Dcxss, the fix ratio for Generic, Provider, and TTP channel was between 30% and 40%, while the WHOIS group achieved a fix rate of less than 20%. In total, fixing these vulnerabilities typically took longer, which stems from the lack of a readily available patch for the custom flaws.

Additionally, as indicated by the increase in viewed reports right after our reminders, we find that they are necessary and useful. Moreover, considering that a patch typically only occurred within the first five days of a report being viewed, *future work should select a shorter interval for such reminders*.

Results Generality — Our work was a first glimpse into the landscape of vulnerability notifications. We explicitly studied the effects of such a campaign in the context of Web vulnerabilities which could be verified without interfering with the server’s normal operation, i.e., we did not consider high-impact flaws such as SQL injections or remote code execution. The impact of our notification campaign was statistically significant, but smaller than in other related experiments (e.g., [13, 21]). Judging from their results, the criticality of the discovered flaws may also effect the impact of a notification campaign.

While in principle we could have applied our methodology to other types of flaws, we limited our study specifically to Web vulnerabilities. Contrary to other works in this space, our methodology to find Client-Side XSS gave us the opportunity to notify domain owners of site-specific flaws rather than vulnerabilities which are the same across multiple installations. Also, to the best of our knowledge, no other parties had access to such a data set, and hence, the site administrators were not aware of the flaws before our notification. This data set gave us the opportunity to study the behavior of administrators when notified of previously-unknown vulnerabilities, and to perform a comparative analysis with a data set of known vulnerabilities.

Web vulnerabilities can be attributed to a *domain*, therefore allowing us to use additional anchors to reach administrators, e.g., in comparison to physical devices such as home routers. We nevertheless believe the methodology can be applied to other types of flaws, to determine whether the vulnerability type influences the impact of notifications. Hence, comparing different means of notifying parties for different types of vulnerabilities is an interesting direction for future research.

Even though the WordPress flaws were publicly known beforehand, they had not received media attention such as, e.g., Heartbleed or NTP amplification attacks. The vulnerabilities we disclosed also concerned the application layer rather than the network layer, i.e., needed to be fixed by a large number of disjunct site owners rather than significantly fewer network providers. Investigating these factors therefore is an interesting direction for future research. This opens new research questions such as: *Are campaigns more successful if the vulnerabilities gained attention in the media (such as Heartbleed)? Does it matter who needs to fix the vulnerability, be it a Web site developer, network admins, or end-users?*

8 Related Work

In this section, we relate our study to prior work, reviewing works on vulnerability notifications, large-scale security analysis and an emerging area of disclosing security-relevant information.

Large-Scale Vulnerability Notification — In concurrent work, Li et al. [23] investigated the feasibility of vulnerability notifications for networked systems, i.e., industry control systems, misconfigured IPv6 firewalls, and DDoS amplifiers. Similarly to our work, they discovered that notifications have a positive impact, but the global effect is low and thus unsatisfactory. Contrary to our work, they did not use links to track the reachability of recipients. They did, however, gain insights into how message content influences fix rates. Moreover, from messages received in a survey they handed out to the notified parties, they found that such notifications generally are welcomed by affected parties, underlining the need for future work in this problem space.

Prior to this work, Li et al. [24] investigated how notification of compromised Web sites can improve the time to clean-up from malware infestation. They find that directly communicating with administrators via the Google Webmaster Console increases likelihood of clean-up by 50% and decreases infection lengths by 62%.

Prior to these closely related works, Durumeric et al. [13] conducted a large-scale analysis of the Heartbleed bug. This work showed that large-scale notification may increase the number of patched servers by about 50%. The main differences between this work and our study are in the composition of the data set. The first, important one is the type of flaw: the Heartbleed bug was a very popular, high-impact flaw with outstanding media coverage and its own website. Our data set does not contain flaws of this type, however it contains previously undisclosed XSS vulnerabilities that, to the best of our knowledge, were unknown to the Web site administrator prior to our disclosure. As a result, our data set allows us to study the problem without bias due to popularity. Second, the authors used the network operator abuse contact (retrieved from the IP WHOIS), whereas we use multiple channels. Finally, our study focuses on Web vulnerabilities, and does not target flaws in the Internet infrastructure, e.g., SSL/TLS.

Similarly to the previous work, Kührer et al. [21] reported on the vulnerability disclosure process on a data set of 9 million servers susceptible to becoming unwitting attackers in NTP amplification attacks. The authors reported all flaws using *en masse* well-established channels. This allowed them to remove 90% of the vulnerable servers within 7 weeks. As opposed to our paper, they relied only on two channels, i.e., TTPs and vendors, and they did not consider other possible ones such as domain WHOIS. More importantly, the authors did not perform a comparative analysis between channels, leaving the research questions of our paper unanswered.

Large-Scale Security Analyses — Recently, we have witnessed an increasing number of large-scale security analyses ranging from validation of security testing tech-

niques (e.g., Balduzzi et al. [2], Lekies et al. [22], Doupé et al. [9]) to Internet-wide analyses of insecure behavior (e.g., Durumeric et al. [11], Kührer et al. [21]), which can spot a large number of security issues. While most of the efforts have been expended on tools and analysis techniques, little has been done to address the problem of reporting the discovered issues. For example, Balduzzi et al. [2] tested 5,000 Web sites for HTTP parameter pollution (HPP), discovering a vulnerability in 30% of them. With reference to the disclosure, the authors left the problem unaddressed, only mentioning that they could not reach all Web site owners. (For other examples in the Web domain, please refer to Doupé et al. [9]).

To better support large-scale analysis, new tools have been developed. For example, ZMap [12] can scan the entire IPv4 address space in 45 minutes. ZMap has already been used for Internet-wide analysis. For example, Durumeric et al. [11] used ZMap to study the HTTPS ecosystem, uncovering a variety of issues including certificates with invalid domains and certificate misuse. Similarly, this paper does not address the problem of reaching operators to solve the problem.

Notification of Security-Relevant Information — This paper can be seen as part of an emerging line of research that develops the idea of using notifications of security-relevant information as a security measure. Works in this area studied the distribution of security-relevant information from different angles and with a major focus on malware reports. For example, Cetin et al. [5] studied the role of sender reputation in abuse reports by sending 480 reports to network providers and Web site owners from senders with different reputations. Their study found no evidence that reputation improves cleanup rates, but they observed that, after accessing an online technical report, network providers performed better than Web site owners.

Vasek and Moore [34] looked at the problem from the angle of the quality of the reports, concluding that detailed reports increase the number of cleanups, while reports with minimal details perform better than not sending reports at all. Our paper builds on the results of these works: we did not consider sender reputation as a variable, and we prepare detailed reports.

Canali et al. [3] studied provider diligence by setting up compromised Web sites and notifying them about ongoing malicious activities. Their experiments showed that 64% of complaints were ignored. While this work shows an alarming attitude towards these problems, the size of their data set, 22 providers, makes it hard to generalize to a larger scale. From this point of view, our paper provides a broader view on the issue, including other notification channels and comparative analysis using a control group as a baseline.

9 Conclusion

With the increase of inter-connectivity on the Internet, the magnitude and diversity of large-scale vulnerability incidents will likely rise. We presented our experiences with a large-scale notification process to inform Web site owners about vulnerable Web apps. While our notifications have had a statistically significant impact on the vulnerability remediation, the overall fix rate is unsatisfactory, leaving 74.5% of Web sites exploitable after our month-long experiment.

This naturally begs the question for the potential reasons for the large fraction of unfixed sites. The major cause is the unsolved challenge to reach out to persons who can deploy a fix, such as developers or administrators. Of all contacts that we notified, only 5.8% viewed our vulnerability report. Out of these, 40% fixed the vulnerability within a week. This, but also the ease of fixing the vulnerabilities (in most cases just update WordPress), indicates that the main problem is actually to *disseminate* the vulnerability information.

How do we inform affected parties about vulnerabilities on large scale? Identifying contact points remains the main challenge that has to be addressed by the Internet society, including network providers, CERTs, and registrars. We imagine that this problem could, for example, be tackled by centralized contact databases, more efficient dissemination strategies within hosters/CERTs, or even a new notification channel or trusted party responsible for such notifications. Until we find solutions to the reachability problem, the effects of large-scale notifications are likely to remain low in the future.

Acknowledgements

The authors would like to thank Thomas Schreck and the Spanish CERT for providing insights into the inner workings of CERT organizations. Also, we would like to thank the anonymous reviewers for their helpful comments. In addition, we thank our shepherd Leyla Bilge for her support in improving the paper for the camera-ready version. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA).

References

- [1] Abusix GmbH. Abuse contact database. <https://abusix.com/contactdb.html>, 2016.
- [2] Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, and Engin Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.
- [3] Davide Canali, Davide Balzarotti, and Aurélien Francillon. The role of web hosting providers in detecting compromised websites. In *Proceedings of the 22nd International World Wide Web Conference*, 2013.
- [4] CERT-CC. List of National CSIRTS. <http://www.cert.org/incident-management/national-csirts/national-csirts.cfm>, 2016.
- [5] Orcun Cetin, Mohammad Hanif Jhaveri, Carlos Ganán, Michel van Eeten, and Tyler Moore. Understanding the role of sender reputation in abuse reporting and cleanup. In *Workshop on the Economy of Information Security (WEIS 2015)*.
- [6] Daniel Cid. Brute force amplification attacks against WordPress XMLRPC. <https://blog.sucuri.net/2015/10/brute-force-amplification-attacks-against-wordpress-xmlrpc.html>.
- [7] D. Crocker. Mailbox Names for Common Services, Roles and Functions. RFC 2142 (Proposed Standard), <http://www.ietf.org/rfc/rfc2142.txt>, May 1997.
- [8] L. Daigle. WHOIS Protocol Specification. RFC 3912 (Draft Standard), <http://www.ietf.org/rfc/rfc3912.txt>, September 2004.
- [9] Adam Doupe, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. Fear the EAR: Discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.
- [10] Drupal Security Team. Drupal core - highly critical - public service announcement - PSA-2014-003. <https://www.drupal.org/PSA-2014-003>.
- [11] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. Analysis of the HTTPS certificate ecosystem. In *Proceedings of the 2013 ACM Internet Measurement Conference*, 2013.
- [12] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.
- [13] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li,

- Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of Heartbleed. In *Proceedings of the 2014 ACM Internet Measurement Conference*, 2014.
- [14] Matthew Finifter, Devdatta Akhawe, and David Wagner. An empirical study of vulnerability rewards programs. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.
- [15] FIRST.org, Inc. Forum of Incident Response and Security Teams. <https://www.first.org/>, 2016.
- [16] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, pages 65–70, 1979.
- [17] ICANN Security and Stability Advisory Committee. SAC 023: Is the WHOIS service a source for email addresses for spammers? <https://www.icann.org/en/system/files/files/sac-023-en.pdf>.
- [18] Iniqua. plecost. <https://github.com/iniqua/plecost>, 2016.
- [19] Joomla! [20151206] - Core - Session Hardening. <https://developer.joomla.org/security-centre/639-20151206-core-session-hardening.html>.
- [20] Aaron Jorbin. WordPress 4.4.1 Security and Maintenance Release. <https://wordpress.org/news/2016/01/wordpress-4-4-1-security-and-maintenance-release/>.
- [21] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from hell? reducing the impact of amplification DDoS attacks. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.
- [22] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013.
- [23] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Damon McCoy, Stefan Savage, Michael Bailey, and Vern Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *Proceedings of the 25th USENIX Security Symposium*, 2016.
- [24] Frank Li, Grant Ho, Eric Kuan, Yuan Niu, Lucas Ballard, Kurt Thomas, Elie Bursztein, and Vern Paxson. Remedyng web hijacking: Notification effectiveness and webmaster comprehension. In *Proceedings of the 25th International World Wide Web Conference*, 2016.
- [25] Suqi Liu, Ian Foster, Stefan Savage, Geoffrey M. Voelker, and Lawrence K. Saul. Who is .com?: Learning to parse WHOIS records. In *Proceedings of the 2015 ACM Internet Measurement Conference*.
- [26] Microsoft Corporation. Services for senders and ISPs. <https://mail.live.com/mail/services.aspx>.
- [27] MITRE Corporation. Common Vulnerabilities and Exposures. <http://cve.mitre.org/>.
- [28] Mutual Internet Practices Association. DomainKeys Identified Mail. <http://www.dkim.org/>, 2016.
- [29] OpSecAdmin. Operations Security Trust. <https://www.ops-trust.net/>, 2016.
- [30] Karl Pearson. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50 (302):157–175, 1900.
- [31] SPF Project. Sender Policy Framework. <http://www.openspf.org/>, 2016.
- [32] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, 2015.
- [33] Team Cymru. Team Cymru IP to ASN Mapping. <http://www.team-cymru.org/IP-ASN-mapping.html>, 2016.
- [34] Marie Vasek and Tyler Moore. Do malware reports expedite cleanup? An experimental study. In *5th Workshop on Cyber Security Experimentation and Test, CSET*, 2012.
- [35] W3Techs. Usage of content management systems for websites. http://w3techs.com/technologies/overview/content_management/all/.

A Notification Email

Subject: Vulnerability Notification for your domain
...com

Hello,

Primer: All information in and attached to this email is confidential and should be passed on to individuals and organizations on a need-to-know principle only.

We are security researchers from Saarland University, Germany. In our research, we have been scanning several web sites for critical vulnerabilities. We would like to inform you that your website is susceptible to the following vulnerability(ies):

- XMLRPC Multicall Vulnerability

The XMLRPC API of Wordpress can be abused to execute numerous commands on the server-side, thereby allowing an attacker to bruteforce passwords or perform a Denial-of-Service attack against the server.

You can review more detailed information using our web interface at [https://notify.mmcii.uni-saarland.de/....](https://notify.mmcii.uni-saarland.de/). Alternatively, you can retrieve more information via email. To do so, please respond to this email and set the subject line to *only* contain the token 72cf.... We will automatically respond with the vulnerability report via email.

Since this notification is part of an ongoing research project, we will re-scan your web site to see if the vulnerability has been fixed. If you wish us to stop scanning your web site, please contact us at contact@notify.mmcii.uni-saarland.de. Should you need further information or have any other questions, please do not hesitate to contact us using the same email address.

Best Regards,
Ben Stock, Researcher at CISPA

Center for IT-Security, Privacy, and Accountability
Saarland University, Building E9 1
Phone +49 681 302 57377

B Multicall Checker

The Multicall checker uses a timing side-channel to determine whether a WordPress XMLRPC service is vulnerable or not. First, we estimate network latency with a probe request. We use the round-trip time of the probe to rule out network transmission time from the measurement of the multicall request. The resulting value is an estimation of the CPU time. However, as this value may be sensitive to fluctuations of the network latency estimation, we increase the execution time with inefficient user credentials. These strings trigger WordPress sanitization procedures, resulting in longer execution time,

and guarantee to always fail at login attempts to prevent accidental unauthorized access to a WordPress installation. Our test estimates CPU time of two multicall requests with 40 and 80 calls per request. We measured that our servers take 1,600 ms and 3,300 ms of CPU time on an Intel i7 processor for 40 and 80 multicalls, respectively, which we use as indicators for vulnerable services. Finally, we correlate the time analysis with the version of the deployed WordPress, which can be extracted from several sources. If both timing analysis and version indicate a vulnerable service, then we mark the Web site as *Exploitable*, otherwise as *Non-Exploitable*.

C Confidence Function

The monitoring system for WordPress returns a collection of time series $T_{ws}^v = s_1 \cdot s_2 \cdot \dots \cdot s_n$ where ws is the Web site, v a vulnerability, and s_i is the result of the checker in a point of time i with $1 \leq i \leq n$. The value s_i can be E if the checker found the vulnerability *Exploitable*; N for *Non-Exploitable*. In our experiments, we need to establish whether a Web site ws has fixed v . To do that, we take into account the rate of unlikely events within a time series in order to establish a confidence level, i.e., the number of substrings $N \cdot E$ in the longer prefix of the time series. We define the confidence that the Web site is not vulnerable as the complement of this rate. We define a Web site as not vulnerable if the confidence is greater than 0.99. If no errors occur in a time series, we define a conservative error of 0.1, i.e., a domain is marked fixed if it was *Non-Exploitable* for three consecutive days.

D Comparison of Notified Groups

	Generic	WHOIS	Provider	TTP
Generic	-	0.0323810	0.2982599	0.2456719
WHOIS	0.0323810	-	0.2714901	0.3276381
Provider	0.2982599	0.2714901	-	0.9038795
TTP	0.2456719	0.3276381	0.9038795	-

Table 7: p -values from χ^2 tests for D_{WP}

	Generic	WHOIS	Provider	TTP
Generic	-	0.5285343	0.1360734	0.3841099
WHOIS	0.5285343	-	0.3862386	0.1346949
Provider	0.1360734	0.3862386	-	0.0191932
TTP	0.3841099	0.1346949	0.0191932	-

Table 8: p -values from χ^2 tests for D_{CXSS}

You've Got Vulnerability: Exploring Effective Vulnerability Notifications

Frank Li[†] Zakir Durumeric^{*‡*} Jakub Czyz^{*} Mohammad Karami[◊]
Michael Bailey[‡] Damon McCoy[▷] Stefan Savage[○] Vern Paxson^{†*}

[†]*University of California Berkeley* ^{*}*University of Michigan* [◊]*George Mason University*

[‡]*University of Illinois Urbana-Champaign* [▷]*New York University*

[○]*University of California San Diego* ^{*}*International Computer Science Institute*

Abstract

Security researchers can send vulnerability notifications to take proactive measures in securing systems at scale. However, the factors affecting a notification's efficacy have not been deeply explored. In this paper, we report on an extensive study of notifying thousands of parties of security issues present within their networks, with an aim of illuminating which fundamental aspects of notifications have the greatest impact on efficacy. The vulnerabilities used to drive our study span a range of protocols and considerations: exposure of industrial control systems; apparent firewall omissions for IPv6-based services; and exploitation of local systems in DDoS amplification attacks. We monitored vulnerable systems for several weeks to determine their rate of remediation. By comparing with experimental controls, we analyze the impact of a number of variables: choice of party to contact (WHOIS abuse contacts versus national CERTs versus US-CERT), message verbosity, hosting an information website linked to in the message, and translating the message into the notified party's local language. We also assess the outcome of the emailing process itself (bounces, automated replies, human replies, silence) and characterize the sentiments and perspectives expressed in both the human replies and an optional anonymous survey that accompanied our notifications.

We find that various notification regimens do result in different outcomes. The best observed process was directly notifying WHOIS contacts with detailed information in the message itself. These notifications had a statistically significant impact on improving remediation, and human replies were largely positive. However, the majority of notified contacts did not take action, and even when they did, remediation was often only partial. Repeat notifications did not further patching. These results are promising but ultimately modest, behooving the security community to more deeply investigate ways to improve the effectiveness of vulnerability notifications.

1 Introduction

A secure Internet ecosystem requires continual discovery and remediation of software vulnerabilities and critical misconfigurations. Security researchers discover thousands of such issues each year, across a myriad of platforms [1]. This process consists of four key phases: (1) discovering new security problems, (2) identifying remedies, (3) determining affected parties, and (4) reaching out to promote remediation among those affected.

The security community has decades of experience with the first two phases, and developments in high-speed scanning [10, 11] and network monitoring [23, 25] have significantly advanced the ease of the third phase for many security issues. However, the process of outreach remains today at best ad hoc. Unlike the public health community, which has carefully studied and developed best practices for patient notification (e.g., [4, 19]), the security community lacks significant insight into the kinds of notification procedures that produce the best outcomes.¹ Instead, for most software, the modern practice of vulnerability notification remains broadcasting messages via well-known mailing lists or websites that administrators must periodically poll and triage.

Given the relative ease with which investigators can today often determine the affected parties, the question then arises of how they should best utilize that information. In the past, performing large-scale notifications was often seen as both ineffective and impractical [2, 7, 12, 18]. However, several recent case studies have provided clear evidence to the contrary. For example, to promote patching of the 2014 OpenSSL Heartbleed vulnerability, Durumeric et al. emailed notices to operators of hosts detected as vulnerable via scanning

¹An exception concerns the development of online software update systems that explicitly tie together notification and remediation, allowing precise and automated updating targeting the affected parties. Unfortunately, the vast majority of software lacks such systems; even for those that do, operators may disable it in some contexts (critical servers, embedded systems) to avoid unplanned downtime.

and found that notified operators patched at a rate almost 50% greater than a control group [9]. Similarly, Li et al. analyzed the efforts of Google Safe Browsing and Search Quality in reaching out to operators of compromised websites, and found that direct communication with webmasters increased the likelihood of cleanup by over 50%, and reduced infection durations by more than 60% [14].

With these clear indications that notifications *can* drive positive security outcomes, it behooves the security community to determine how to best conduct the outreach efforts. At the same time, we must balance the benefits to the ecosystem (and the associated ethical responsibilities to notify) against the burden this imposes on the reporter, which calls for determining notification regimens that will not prove unduly taxing.

In this work, we strive to lay the foundations for systematically determining the most effective notification regimens, seeking to inform and drive the development of “best practices” for the community. The solution space has many more dimensions than we can hope to methodically explore in a single study. Here, we aim to develop soundly supported results for the most salient basic issues, with an eye towards then facilitating follow-on work that builds on these findings to further map out additional considerations. The issues we address include (1) who to notify (e.g., WHOIS contacts versus national CERTs versus US-CERT), (2) the role of notification content (e.g., do reporters need to devise detailed messages or do short ones suffice), (3) the importance of localization (e.g., what role does native language play in notification response rates), and (4) how these considerations vary with the nature of the vulnerability (including whether for some vulnerabilities notification appears hopeless).

We evaluate these questions empirically in the context of notification campaigns spanning three different vulnerability categories: publicly accessible industrial control systems, misconfigured IPv6 firewalls, and DDoS amplifiers. Using large-scale Internet scanning to identify vulnerable hosts and then monitor their behavior over time post-notification, we infer the effects of different notification regimes as revealed by the proportion and timeliness of contacts remediating their vulnerable hosts.

Our results indicate that notifications can have a significant positive effect on patching, with the best messaging regimen being directly notifying WHOIS contacts with detailed information within the message itself. An additional 11% of contacts addressed the security issue when notified in this fashion, compared to a control. However, we failed to push the majority of contacts to take action, and even when they did, remediation was often only partial. Repeat notifications did not further patching. We additionally characterize the responses we

received through our notification campaigns, of which 96% of human-sent responses were positive or neutral. Given these promising yet modest findings, it behooves the security community to more deeply investigate vulnerability notifications and ways to improve their efficacy. Our methodology and results form the basis for establishing initial guidelines to help drive future efforts.

2 Related Work

Several recent studies have found that large-scale security notifications increase patching and remediation—particularly for infected websites.

Vasek et al. notified 161 infected websites [24] and found that after 16 days, 55% of notified sites cleaned up compared to 45% of unnotified sites. They further note that more detailed notifications outperformed reports with minimal information by 13%, resulting in a 62% cleanup rate. Cetin et al. performed a similar study, measuring the role of sender reputation when notifying the owners of hijacked websites [5]. They emailed the WHOIS contacts of 240 infected sites from email addresses belonging to an individual independent researcher (low reputation), a university research group (medium reputation), and an anti-malware organization (high reputation). While nearly twice as many notified sites cleaned up within 16 days compared to unnotified ones, they found no significant differences across the various senders.

On a larger scale, Li et al. investigated the life cycles of 761 K website hijacking incidents identified by Google Safe Browsing and Search Quality [14]. They found that direct notifications to webmasters increased the likelihood of cleanup by over 50% and reduced infection lengths by 60% on average. Absent this communication, they observed that browser interstitials—while intended to protect browser users—correlated with faster remediation.

Most similar to the vulnerabilities we investigate, Durumeric et al. used Internet-wide scanning to track the Heartbleed vulnerability and notified system owners two weeks after public disclosure [9]. Their notifications drove a nearly 50% increase in patching compared to a control: 39.5% versus 26.8%.

Concurrent to this work, Stock et al. investigated the feasibility of large-scale notifications for web vulnerabilities [22]. Similar to our study, they experimentally evaluated the effectiveness of different communication channels, including WHOIS email contacts and CERTs. Additionally, they analyzed the reachability and viewing behavior of their messages. Their results largely accord with ours, providing a complementary study of notifications in a separate context (namely, vulnerable websites). Notably, they likewise observed that while notifications

Dataset	Hosts	WHOIS Abuse Contacts	Hosts with WHOIS Contacts
ICS	45,770	2,563	79.7%
IPv6	180,611	3,536	99.8%
Ampl.	83,846	5,960	92.4%

Table 1: Vulnerable Hosts—We notified network operators about three classes of vulnerabilities found in recent studies: publicly accessible industrial control systems (ICS), hosts with misaligned IPv4 and IPv6 firewall policies, and DDoS amplifiers (NTP, DNS, and Chargen).

could induce a statistically significant increase in patching, the raw impact was small. In the best case, only an additional 15% of the population patched compared with a control group.

Each of these studies has established that notifications can increase vulnerability patching and cleanup. We build on these works and explore the next critical step: understanding what factors influence patching and how to construct effective vulnerability notifications.

3 Methodology

To measure notification efficacy and to understand how to construct effective notifications, we notified network operators while varying aspects of the notification process. In this section, we detail the datasets of vulnerable hosts, the variables we tested, and how we tracked remediation.

3.1 Vulnerable Hosts

We notified operators about the three classes of vulnerabilities listed below. We show the population of vulnerable hosts in Table 1.

Publicly Accessible Industrial Control Systems Industrial control systems (ICS) are pervasive and control physical infrastructure ranging from manufacturing plants to environmental monitoring systems in commercial buildings. These systems communicate over a myriad of domain and manufacturer specific protocols, which were later layered on Ethernet and TCP/IP to facilitate long distance communication. Never designed to be publicly accessible on the Internet, these protocols lack important security features, such as basic authentication and encryption, but nonetheless are frequently found unsecured on the public Internet. To identify vulnerable ICS devices, Mirian et al. extended ZMap [10] and Censys [8] to complete full IPv4 scans for several ICS protocols: DNP3, Modbus, BACnet, Tridium Fox, and Siemens S7 [17]. In total, they found upwards of

46 K ICS hosts that were publicly accessible and inherently vulnerable.

We coordinated with Mirian et al. to complete daily scans for each protocol against the public IPv4 address space from January 22–24, 2016. We limited our study to the 45.8 K hosts that were present all three days to reduce the noise due to IP churn. To track the impact of our notifications, we continued the daily scans of these hosts using the same methodology.

Misconfigured IPv6 Firewall Policies Czyz et al. found that 26% of IPv4/IPv6 dual-stack servers and routers have more permissive IPv6 firewall policies compared to IPv4, including for BGP, DNS, FTP, HTTP, HTTPS, ICMP, MySQL, NTP, RDP, SMB, SNMPv2, SSH, and Telnet access [6]. For example, twice as many routers have SSH accessible over IPv6 compared to IPv4. Given the presumed rarity of IPv6-only services, this likely indicates a misconfiguration and potential security issue.

To identify dual-stack servers, Czyz et al. looked for hostnames in the Rapid7 DNS ANY dataset [20] that had both A and AAAA records. After filtering out automatically generated hostnames, they identified 520 K dual-stack servers. To find routers, the team performed reverse DNS lookups and subsequent A and AAAA lookups for hosts in the CAIDA Ark dataset [3], identifying 25 K routers. Czyz et al. then scanned these hosts using Scamper [15] to identify firewall inconsistencies.

We scanned the hosts that Czyz et al. identified over a 25 day period from December 31, 2015 to January 24, 2016. We limited our study to the 8.4 K routers and 172.2 K servers that were consistently available during that period. Similar to the ICS measurements, we continued to perform daily scans using the same methodology to track the impact of our notifications.

DDoS Amplifiers Several UDP protocols allow attackers to launch distributed denial of service attacks when improperly configured [21]. In this scenario, an attacker spoofs a small request to a misconfigured server, which then sends a large response to the victim. For example, an attacker can spoof a DNS lookup to a recursive DNS resolver, which will then send the full recursive lookup to the victim’s machine. We identified 152 K misconfigured hosts that were actively being used to launch DDoS attacks over NTP, DNS, and Chargen by monitoring the sources of DDoS attacks against a university network between December 11–20, 2015.

We restricted our notifications to the vulnerable hosts that were consistently available during our daily scans from December 21, 2015 to January 26, 2016. In total, we discovered 5.9 K Chargen amplifiers, 6.4 K NTP amplifiers, and 71.5 K DNS amplifiers on 83.8 K distinct IP addresses. We continued to track these hosts by performing

ing daily protocol scans (e.g., Chargen requests, NTP monlist commands, and DNS recursive lookups).

In each case, we coordinated with the studies’ authors to ensure that they did not simultaneously notify operators. However, we do note that groups have previously sent notifications to DDoS amplifiers [13].

3.2 Experiment Variables

To understand how to best construct and route notification messages, we performed notifications using several methodologies and measured the differences in remediation. We specifically aimed to answer the following questions:

Who should researchers contact? Researchers have several options when deciding where they should report vulnerabilities, including directly contacting network operators, notifying national CERTs, and asking their own country’s CERT to disseminate the data to other CERT groups. We tested three options: (1) notifying the abuse contact from the corresponding WHOIS record, (2) geolocating the host and contacting the associated national CERT, and (3) asking our regional CERT (US-CERT) to propagate the information.

How verbose do messages need to be? It is not clear how much information researchers need to include when notifying operators. For example, are notifications more effective if researchers include detailed remediation steps or will such instructions go unheeded? We sent three types of messages: (1) a terse message that briefly explained that we discovered the vulnerability with Internet-wide scanning, and the impact of the vulnerability (e.g., for ICS notifications, we wrote “These devices frequently have no built-in security and their public exposure may place physical equipment at risk for attack.”), (2) a terse message with a link to a website with detailed information, and (3) a verbose email that included text on how we detected the problem, vulnerability details, and potential remediation steps. We provide the full text of our different messages in Appendix B–G.

Do messages need to be translated? We tested sending messages in English as well as messages translated by native technical speakers to several local languages.

3.3 Group Assignment

To test the impact of our experiment variables, we randomly formed experiment groups that received different notification regimens. Here we describe our process for constructing these groups.

For each IP address, we extracted the abuse contact from the most specific network allocation’s WHOIS

Group	ICS	IPv6	Ampl.
Control	657	3,527	1,484
National CERTs	174	650	379
US-CERT	493	578	1,128
WHOIS: English Terse	413	633	777
WHOIS: English Terse w/ Link	413	633	777
WHOIS: EnglishVerbose	413	632	777
WHOIS: Language – Terse			
Germany: German		71	
Germany: English		72	
Netherlands: Dutch		32	
Netherlands: English		32	
Poland: Polish		37	
Poland: English		37	
Russia: Russian		123	
Russia: English		123	
WHOIS: Language –Verbose			
Germany: German	70		
Germany: English	72		
Netherlands: Dutch	32		
Netherlands: English	29		
Poland: Polish	36		
Poland: English	36		
Russia: Russian	123		
Russia: English	123		

Table 2: Notification Groups—We aggregated vulnerable hosts by WHOIS abuse contacts and randomly assigned these contacts to notification groups. Here, we show the number of contacts notified in each group. Note that for the language experiments, we tested terse and verbose messages for several countries, both translated and in English.

record. For the 16.7% of dual-stack hosts with different contacts extracted from IPv4 and IPv6 WHOIS records, we used the contact with the deepest level of allocation, and preferred IPv6 contacts when all else was equal (4.3% of dual-stack hosts).

To test each variable, we split the abuse contacts from each vulnerability into treatment groups (Table 2). For the ICS and amplifier experiments, we randomly allocated one quarter of abuse contacts to the control group (Group 1), one quarter to the CERT groups (half US-CERT, half national CERTs), and the remaining half to the WHOIS groups. For IPv6, to act in a responsible manner we needed to complete *some* form of notification for all hosts to ensure adequate disclosure prior to the release of the corresponding study [6] in February 2016. This prevented us from using a true control group. Instead, we approximate the behavior of the control group using the 25 days of daily scans prior to our notifications. We allocated a third of the IPv6 contacts to the CERT groups, and the remainder to the WHOIS groups.

For the vulnerable hosts assigned to the CERT groups,

we geolocated each IP using MaxMind [16] and identified the associated CERT. We note that not all countries have an established CERT organization. This was the case for 2,151 (17%) IPv6 hosts, 175 (8%) ICS devices, and 2,156 (19%) DDoS amplifiers. These hosts were located in 16 countries for IPv6, 26 countries for ICS, and 63 countries for DDoS. Many of these countries are in Africa or Central America (e.g., Botswana, Ethiopia, and Belize), or are smaller island states (e.g., American Samoa, Antigua and Barbuda, and the Bahamas). We did not include hosts without a CERT organization in the CERT experiment (although we later passed them along to US-CERT).

In total, 64 CERTs were responsible for IPv6 hosts, 57 for ICS, and 86 for amplifiers. To compare directly contacting national CERTs versus having US-CERT distribute information to them, we randomly divided the affected national CERTs into two halves. For national CERTs in the first half, we contacted them directly with vulnerable hosts in their region (Group 2). We sent the remaining hosts for CERTs in the second half to US-CERT (Group 3).

We obtained native translations of our WHOIS messages for several countries. We allocated contacts in the WHOIS groups that were in those countries (based on the WHOIS records) for our language experiment, further detailed in Section 4.3. The remaining contacts were randomly split into three groups based on message verbosity: terse (Group 4), terse with a link (Group 5), and verbose (Group 6).

3.4 Notification Process

We sent notification emails with the FROM and REPLY-TO header set to an institutional mailing list: *security-notifications@berkeley.edu*. In each message, we attached a CSV file that contained the list of vulnerable hosts along with the latest scan timestamp and the list of vulnerable protocols. We also included a link to an anonymous survey, which asked for the organization’s perspective on the reported security issue and whether they found our detection and notification acceptable. The messages were sent from a server in UC Berkeley’s network, which was listed as a valid mail server by UC Berkeley’s SPF policy. We note that we also included a randomly generated identifier in each email subject that enabled us to match a reply to the original notification.

3.5 Tracking Remediation

We tracked the impact of different notification methodologies by scanning all hosts for several weeks following our notifications. As our scanning methods tested the reachability of several services, we may have falsely

identified a host as patched due to random packet loss or temporary network disruptions. To account for this, we only designated a host as patched if it did not appear vulnerable in any subsequent scans. We leveraged the last day’s scan data for this correction, but did not otherwise use it in our analysis as it lacked subsequent data for validation.

One limitation in our tracking is the inability to distinguish true patching from network churn, where the host went offline or changed its IP address. While we can still conduct a comparative analysis against our control group, we acknowledge that our definition of patching is a mixture of true patching and churn. We investigated whether we could better approximate true remediation by distinguishing between RST packets and dropped packets. We compared the proportion of RSTs and drops between our control group and our notified groups two days after notification and two weeks after notification. At both times, we observed nearly identical proportions between the control and notified groups—in all cases less than 20% of hosts sent RST packets. This indicates that RST packets are not a reliable signal for remediation, as most hosts did not send RST packets even when truly fixed.

Unless stated otherwise, we consider a host as having taken remediation steps for a particular vulnerability if any of its affected protocols were detected as fixed. Likewise, we say a notification contact has taken remediation steps if any of its hosts have patched. We define the remediation rate as the percentage of notification contacts that have taken remediation steps. This definition is over contacts rather than hosts as we are measuring the impact of notifying these contacts, and contacts differ in the number of affected hosts.

3.6 Ethical Considerations

We followed the guidelines for ethical scanning behavior outlined by Durumeric et al. [10]: we signaled the benign intent of our scans through WHOIS entries and DNS records, and provided project details on a website on each scanning host. We respected scanning opt-out requests and extensively tested scanning methods prior to their deployment.

The ethics of performing vulnerability notifications have not been widely discussed in the security community. We argue that the potential good from informing vulnerable hosts outweighs the risks. To minimize potential harm, we only contacted abuse emails using addresses available in public databases. Additionally, we messaged all unnotified contacts at the conclusion of the study. We offered a channel for feedback through an anonymous survey with questions about the notified organization (described in Appendix A). We note that be-

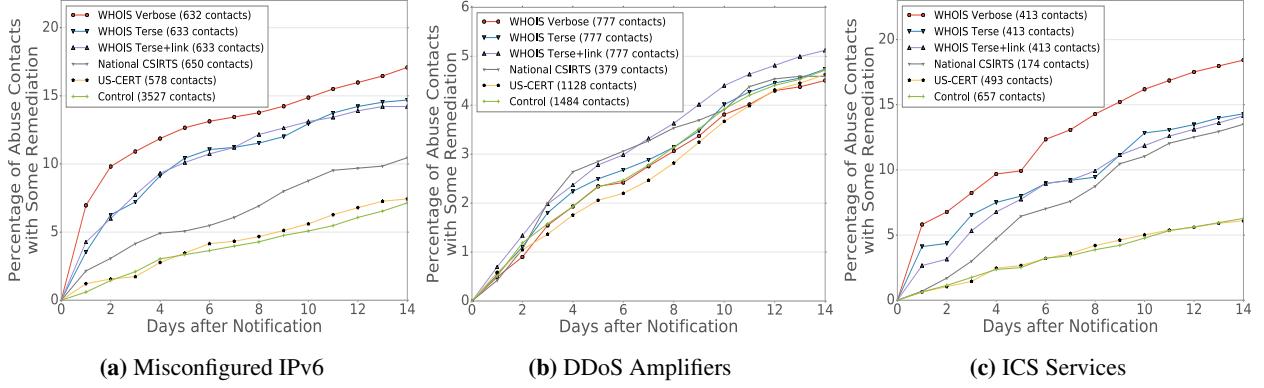


Figure 1: Remediation Rates—We show the remediation rate for each variable we tested. We find that verbose English notifications sent to network operators were most effective for IPv6 and ICS. Note the varying Y axes.

cause we only collected data about organizational decisions and not individuals, our study did not constitute human subjects research (confirmed by consulting the UC Berkeley IRB committee). Nevertheless, we followed best practices, e.g., our survey was anonymous and optional.

4 Results

For both ICS and IPv6, our notifications had a significant impact on patch rates. In our most successful trial—verbose English messages sent directly to operators—the patch rate for IPv6 contacts was 140% higher than in the control group after two weeks. For ICS, the patch rate was 200% higher. However, as can be seen in Figure 1b, none of our notifications had significant impact on DDoS amplifiers. This is likely due to the extensive attention DDoS amplifiers have already received in the network operator community, including several prior notification efforts [21]. In addition, these amplifiers were already previously abused in DDoS attacks without administrative responses, potentially indicating a population with poor security stances. It is also important to note that our best notification regimen resulted in at most 18% of the population remediating. Thus, while notifications can significantly improve patching, the raw impact is limited. In the remainder of this section, we discuss the impact of each experiment variable and how this informs how we should construct future notifications.

To characterize the performance of our trial groups, we measure the area under the survival curve for each group, which captures the cumulative effect of each treatment. To determine if observed differences have statistical significance, we perform permutation tests with 10,000 rounds. In each round of a permutation test, we randomly reassign group labels and recompute the area differences under the new assignments. The intuition is

that if the null hypothesis is true and there is no significant difference between two groups, then this random reassignment will only reflect stochastic fluctuation in the area difference. We assess the empirical probability distribution of this measure after completing the permutation rounds, allowing us to determine the probability (and significance) of our observed values.

All reported p -values are computed via this permutation test. We use a significance threshold of $\alpha = 0.05$, corrected during multiple testing using the simple (although conservative) Bonferroni correction, where each test in a family of m tests is compared to a significance threshold of $\frac{\alpha}{m}$.

Ideally, we would have selected this procedure as part of our original experimental design. Unfortunately, we only identified its aptness *post facto*; thus, its selection could introduce a selection bias, a possible effect that we lack any practical means to assess.

4.1 Notification Contact

For both IPv6 and ICS notifications, directly notifying WHOIS abuse contacts was most effective—particularly early on. Two days after IPv6 disclosure, direct verbose notifications resulted in 9.8% of the population remediating, compared to 3.1% when contacting national CERTs and 1.4% by contacting US-CERT. For ICS, direct notifications promoted 6.8% of the population to patch, more than national CERTs (1.7%) and US-CERT (1.0%). In both cases, direct notifications were notably better than no notifications. As can be seen in Figures 1a and 1c, this gain was persistent. After two weeks, the patch rate of directly notified IPv6 contacts was 2.4 times as high as the control, and three times as high for ICS contacts.

To determine if these observations are statistically significant, we perform permutation tests using the Bonferroni correction. With six treatment groups, the family of

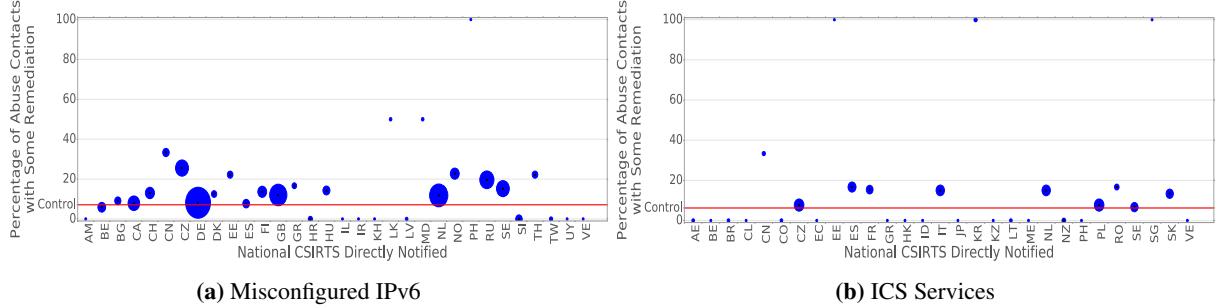


Figure 2: Differences between National CERTs—We show the remediation rate for each directly notified national CERT after two weeks. The size of a data point is proportional to the number of abuse contacts in the country. We directly contacted 32 CERTs for IPv6, and 29 CERTs for ICS. We observe notable differences between CERT groups. However, none are statistically significantly different than the control group. This may be because there are too few hosts for some countries, and that the Bonferroni correction is conservative.

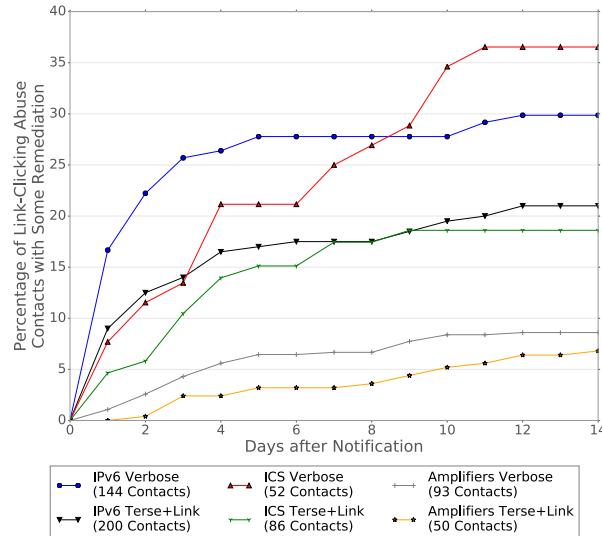


Figure 3: Remediation Rates for Website Visitors—The contacts who viewed our informational website remediated at a higher rate than those who received a verbose message. However, despite this, less than 40% of the contacts who visited the site fixed the vulnerability.

pairwise comparisons includes 15 tests, giving an individual test threshold of $\alpha = 0.0033$. Under the permutation test, the gains that direct verbose notifications had on the CERTs and the control group are statistically significant for both IPv6 and ICS, with all p -values less than 0.0001 except when comparing ICS verbose notifications with national CERTs ($p = 0.0027$).

Notably, US-CERT—our local CERT who we asked to disseminate data to other CERT groups—had the lowest patch rate, which is statistically indistinguishable from the control group that had no notifications. We suspect that US-CERT did not disseminate the data to any other

CERT groups or notify any US operators. One national CERT included in the report to US-CERT informed us they had not received any notices from US-CERT. As seen in Figure 2, there were stark differences between CERT groups—some duly notified operators, while others appear to have ignored our disclosures.

Overall, this suggests that the most effective approach—in terms of both the number of hosts patched and the rate of patching—is to directly notify network operators rather than contact CERT groups.

4.2 Message Verbosity

To determine what information needs to be included in notification messages, we sent three types of emails: (1) verbose, (2) terse, and (3) terse with a link to a website with additional details. We observed the best remediation by contacts who received verbose messages. For IPv6, verbose messages were 56.5% more effective than either terse messages after two days and 55.5% more effective for ICS. However, as can be seen in Figure 1, the differences between verbose and terse messages decreased over time.

Using permutation testing and the Bonferroni correction, we find that the differences between the message types are not statistically significant for IPv6 and ICS. However, given the earlier benefits that verbose messages had for both data sets, we argue notifiers may still want to prefer verbose messages over terse ones. We discuss this effect further in Section 4.4 and note that further investigation of this variable is warranted.

We tracked the remediation rate of contacts who visited the linked website, as shown in Figure 3. We note that all of the information included in the verbose message was available on the linked website and that 16.8% of users who received an email with a link visited the site. This indicates that a sizable population of users engaged

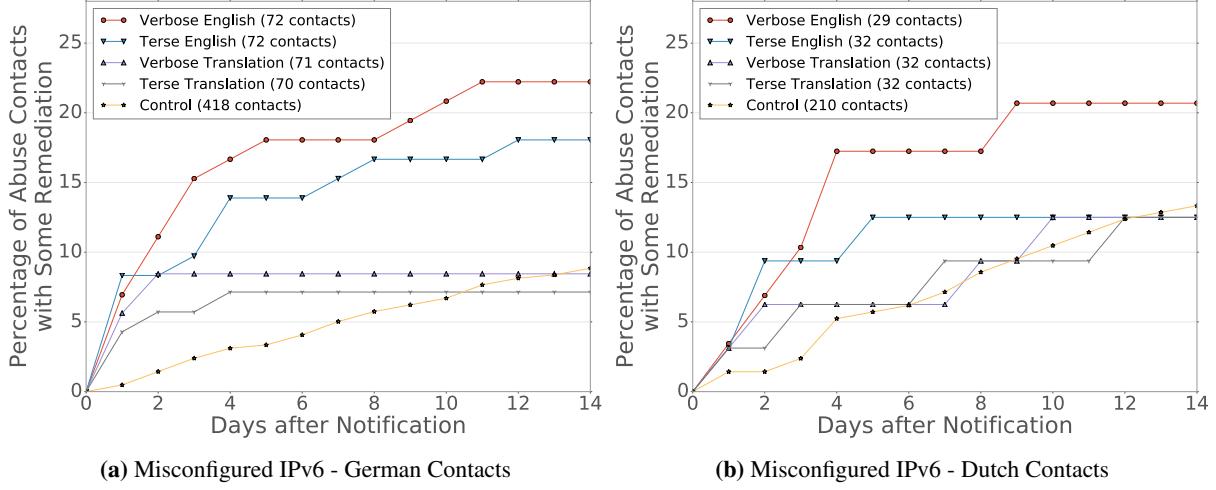


Figure 4: Remediation Rates for Translated Messages—We find that sending verbose English messages was more effective than translating notifications into the language of the recipient. Note, though, that this observation is limited to the small set of languages we were able to evaluate.

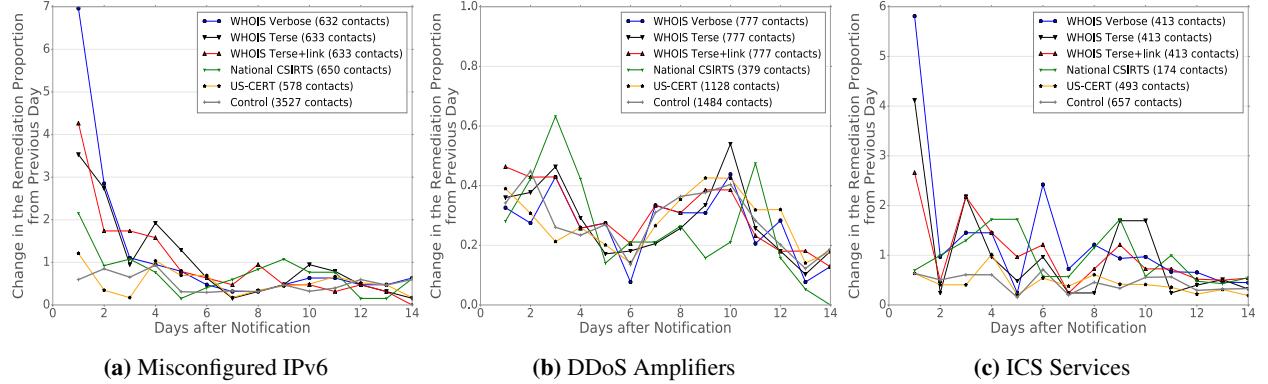


Figure 5: Daily Changes in Remediation Proportions—We show the differences in the proportions of remediated contacts from one day to the next. We find that most contacts that remediated fixed the problem immediately after disclosure. After a few days, contacts returned to remediating at the same rate as the control group.

with our site, but many would not patch even after visiting the link. Specifically, no more than 40% of website visitors patched. Thus, even when our messages successfully reached contacts, the majority did not take action.

4.3 Message Language

To investigate whether notifications need to be translated into recipients’ local languages or can be sent in English, we distributed translated messages for two countries for DDoS and IPv6 notifications. For DDoS amplifiers, we obtained native Russian and Polish translations—for the countries with the third and fourth largest number of vulnerable organizations. For IPv6, we translated messages into German and Dutch, for the second and third largest countries. The population of contacts in non-English

speaking countries for the ICS dataset was too low to provide significant meaning. We randomly split the WHOIS contacts in each country into four groups that vary language and verbosity.

We observe no significant effect from language for DDoS notifications. This is unsurprising given our notifications’ overall lack of effect on DDoS amplifiers. For IPv6, as seen in Figure 4, we observe that translated messages resulted in worse patching than when left in English. Several survey respondents were surprised at receiving translated messages from United States institutions and initially suspected our notifications were phishing messages or spam, which may explain the lower patch rate. The additional overhead of translating messages paired with less successful disclosure suggests that it may be most effective to send notifications in English.

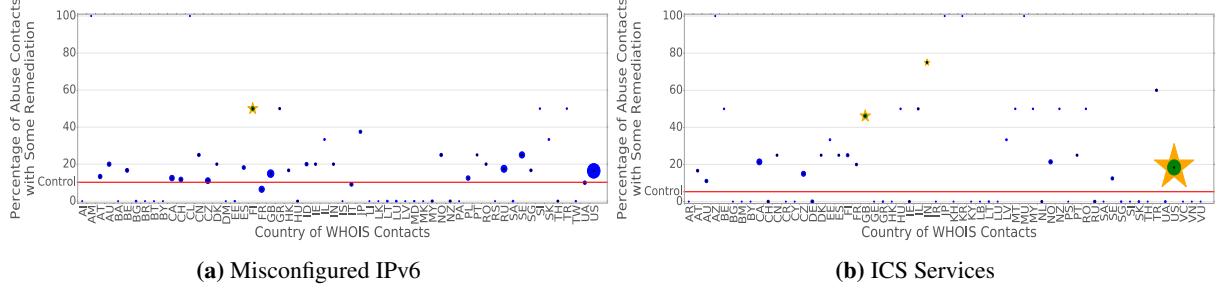


Figure 6: Contact Remediation per Country—We show the percentage of contacts who remediated per country after two weeks. The data sizes are proportional to the number of contacts. Green data points surrounded by an orange star signify countries with a remediation rate statistically better than the control group's, under the permutation test using the Bonferroni correction.

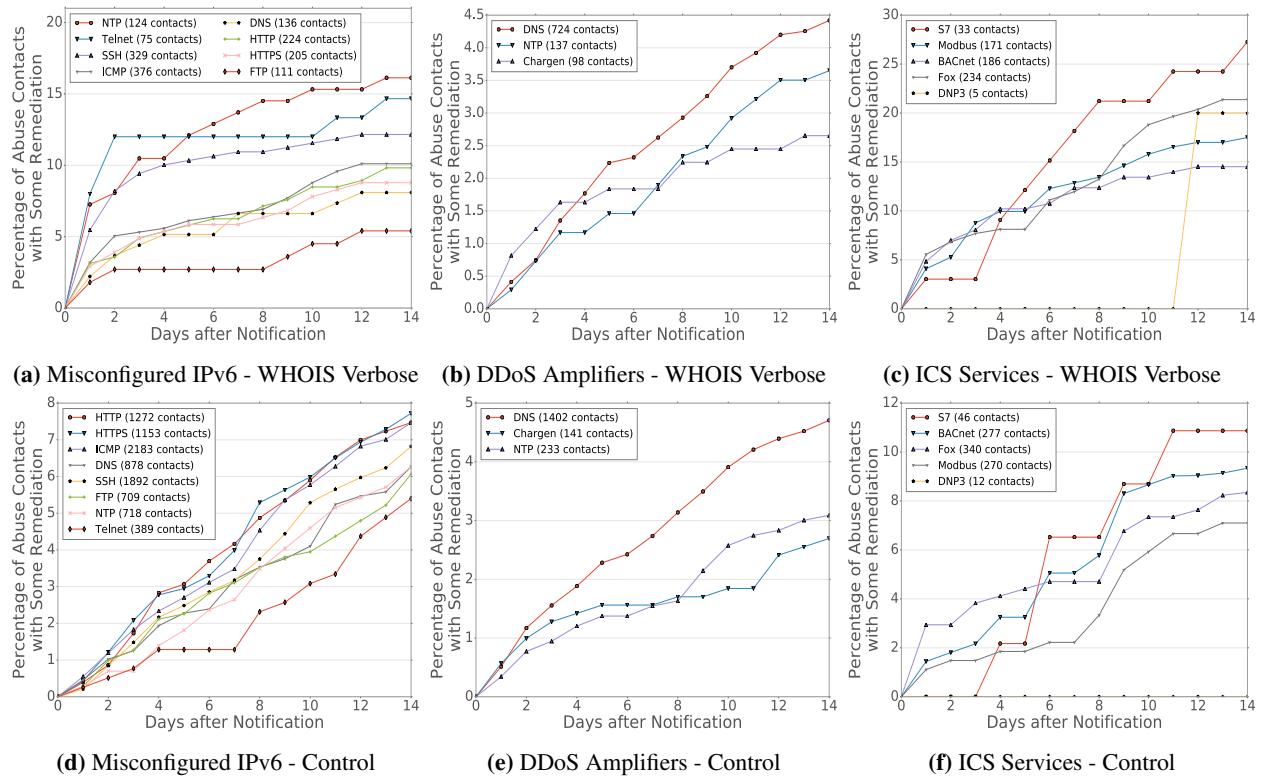


Figure 7: Protocol Remediation Rates—We track the remediation rate for each specific protocol within the WHOIS verbose group and the control group. We note that operators patched some protocols significantly faster than others (e.g., Telnet versus FTP).

However, we note that our results are limited to the small set of languages we were able to obtain reliable translations for, and deeper investigation into the effects of message language is warranted.

4.4 Staying Power of Notification's Effect

As can be seen in Figure 5, our notifications caused a near immediate increase in patching. However, this

increased patching velocity did not persist. In other words, we find that the effects of notifications were short-lived—on the order of several days. The day after notifications were sent, we observe large increases in the remediation proportions for IPv6 and ICS notified groups, as operators responded to our reports. However, we also see that the daily changes in remediation proportions drastically dropped by the second day.

For IPv6, the daily changes in remediation proportions

for all notified groups leveled off and matched that of the control group from the fifth day onward. We also witness a drop off in the daily remediation proportion changes for ICS, although a non-trivial amount of change continued throughout the first 10 days. Notably, the national CERTs first began accelerating remediation after two days, a delay compared to WHOIS experiment groups. For amplifiers, there was little change in the remediation rate over time, which is unsurprising given the limited effect of our notifications.

4.5 Geographic Variation

As with the national CERTs, we note variation in the patching rates between countries. This suggests that the geographic distribution of vulnerable contacts may influence a notification’s outcome. As visible in Figure 6, the United States, Great Britain, India, and Finland were the only countries that patched significantly better than the control group. However, we note that some countries had too few hosts to be statistically significant, given the conservative nature of the Bonferroni correction.

4.6 Variation over Protocols

In Figure 7, we observe variation in the patch rates for different protocols within each vulnerability (e.g., Modbus versus S7 for ICS). As seen in Figure 7a, network administrators reacted most to open IPv6 NTP, Telnet, and SSH services, and least to FTP, with over a 200% difference in the remediation proportions. This variation is not reflected in the control group (Figure 7d), where all protocols exhibited similar behavior. This may reflect an increased likelihood that certain services were unintentionally left accessible, or that operators assessed different levels of risk for allowing different protocols to be reachable.

Operators also responded differently for the multiple ICS protocols (Figure 7c), but the variation is also reflected for contacts in the control group (Figure 7f). BACnet, Fox, and Modbus devices were fixed at similar rates. While the remediation of S7 systems initially lagged behind, there was a significant upswing in action after three days, with nearly 18% of contacts with vulnerable S7 systems patching after 8 days.

Surprisingly, no DNP3 systems had been patched within 10 days of notification (out of 5 contacts). We note that these five contact groups belonged to Internet service providers—not individual organizations. We similarly note that DNP3 differs from the other protocols and is specifically intended for power grid automation. These devices may be remote power stations which require more complex changes than local devices

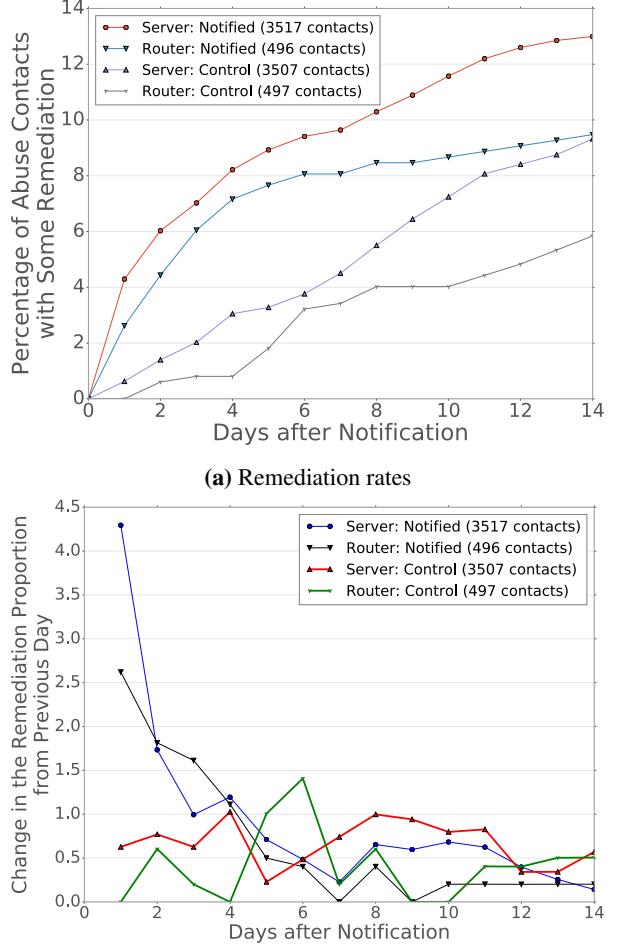


Figure 8: Remediation Rates by Host Type—We find no significant difference in the remediation rate between servers and routers.

(e.g., installation of new hardware versus a configuration change).

While we observe variation between amplifier protocols, these fluctuations are similar in both the notified and control group. Given the limited effect of our DDoS amplifier notifications, these differences likely reflect the varying natural churn rates of these hosts.

4.7 Host Type

When notifying IPv6 operators, we were able to distinguish between servers and routers. To assess the difference between device types, for each type, we only consider contacts with a vulnerable host of that type. We count a contact as having performed some remediation if that contact fixed at least one host of that type.

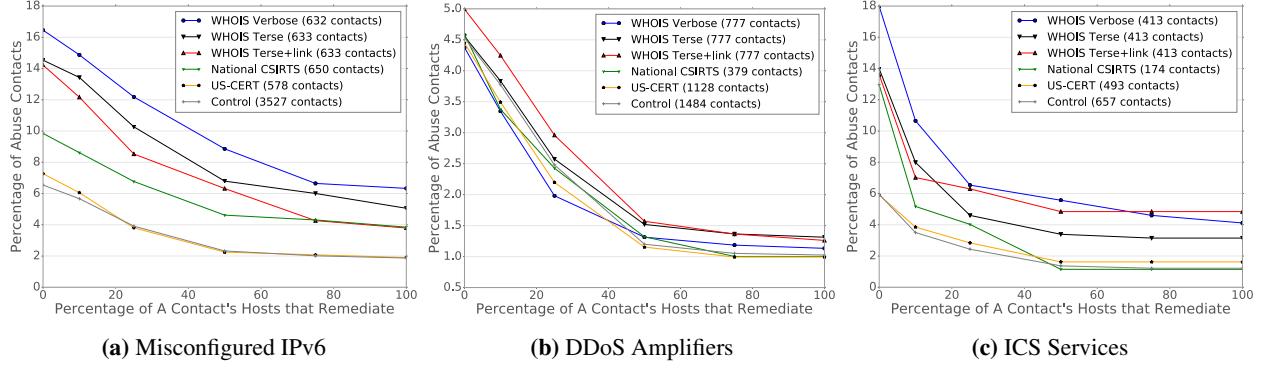


Figure 9: Remediation Completeness—We find that most operators only fixed a subset of their vulnerable hosts. For example, only 40% of the operators that fixed a single host fixed all hosts in their purview.

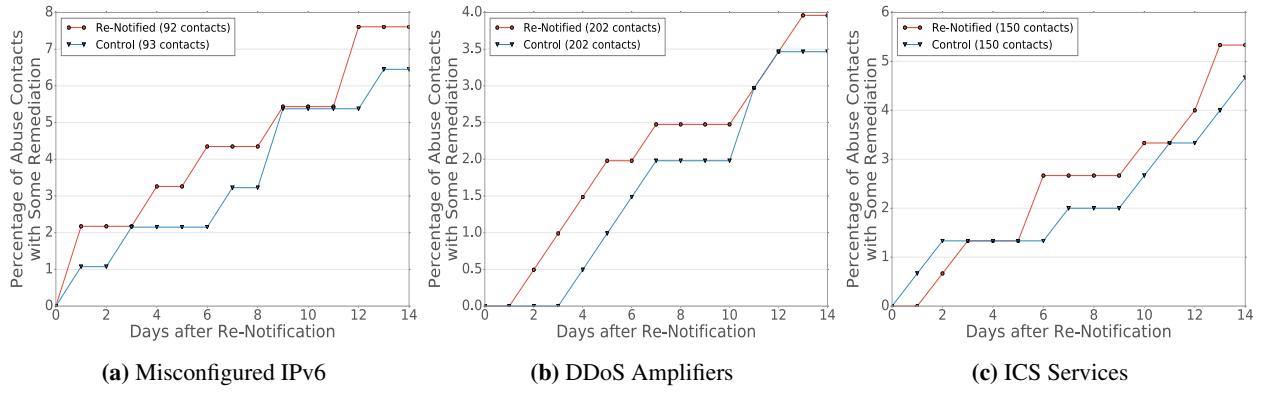


Figure 10: Re-Notifications—We find that a second round of notifications did not result in increased remediation.

We observe that servers and routers remediated at similar rates for the first four days, after which router remediation dropped off and fell significantly below that of servers (Figure 8a). However, servers also naturally patched at a higher rate than routers in the control group. This difference accounts for the gap between notified servers and routers after four days. This is also visible in Figure 8b, where the daily changes in the remediation proportions converged after four days. After 14 days, notified contacts with servers fixed at a rate 44% higher than notified contacts with routers. The divergence in the control group was similar at 48%. This indicates that overall, network administrators respond to vulnerabilities in servers and routers about equally.

4.8 Degree of Remediation

Up to this point, we designated a contact as having patched if any host under its purview was patched. We now consider how well operators patched their hosts.

As can be seen in Figure 9, the majority of contacts did not patch all of their servers. Less than 60% secured all hosts and we note that 30% of groups with 100% reme-

diation were only responsible for fixing one or two hosts. This highlights one of the challenges in the vulnerability notification process: even if our messages reach a designated contact, that contact may not have the capabilities or permissions to remediate all hosts. The multiple hops in a communication chain can be broken at any link.

4.9 Repeated Notifications

Given that our notifications resulted in improved patching, a natural question is whether repeat notifications promote further remediation. We conducted a second round of notifications for the contacts that were directly sent verbose messages in the first round since these proved to be the most effective. We randomly split contacts who had not remediated one month after our notifications into two groups, one as a control group and one to receive a second round of notifications.

As can be seen in Figure 10, the patch rates between the re-notified group and the control group were similar for all three vulnerabilities, indicating that repeat notifications are not effective. This suggests that contacts who did not remediate during the first round of notifications

either were not the appropriate points of contact, or chose (either intentionally or due to lack of capabilities) to not remediate. It is unlikely they simply missed or forgot about our original notification.

5 Notification Reactions

We included a link to an anonymous survey in all of our notification emails as well as monitored the email address from which we sent messages. In the two weeks following our disclosures, we received 57 survey submissions and 93 human email replies. In this section, we analyze these responses.

5.1 Email Responses

Of the 685 email responses we received, 530 (77%) were automated responses (e.g., acknowledgment of receipt), 62 (9%) were bounces, and 93 (14%) were human responses (Table 3). For all three vulnerabilities, over 70% of the human responses expressed positive sentiments. We received only four negative emails, all of which concerned IPv6. Two stated that we were incorrectly using the abuse contact; the other two noted that the open IPv6 services were intentional and asked to be excluded from notifications in the future. None of the emails were threatening. We detail the breakdown for each vulnerability type in Table 4.

Beyond expressing sentiments, 23 contacts requested additional information—primarily about how we detected the vulnerabilities; two requested remediation instructions. Of those 23 contacts, 15 (65%) received terse notifications without a link to additional information, while 3 contacts (13%) received verbose messages. We note that verbose messages both reduced follow-up communication and resulted in the highest patching rate.

Unexpectedly, all five contacts who requested information about DDoS amplifiers asked for evidence of DDoS attacks via network logs. This may be a result of the extensive attention amplifiers have received in the past, such that operators only respond to active abuse issues regarding amplifiers.

Twelve IPv6 contacts rebutted our claim of vulnerability. Six stated that the inconsistency was intentional; one was a honeypot; and five explained that the IP addresses we sent them no longer pointed to the same dual-stack host, likely due to network churn. Two amplifier contacts claimed we falsely notified, stating that their hosts were honeypots. However, we do note that these IPs were seen as part of an attack and were therefore likely misconfigured honeypots.

Most human responses were in English, with eight (9%) in other languages: 3 Russian, 1 German, 1 Czech, 1 Swedish, 1 French, and 1 Slovak. These non-English

Response Types	ICS	IPv6	Ampl.
Automated	143	214	173
Human	22	48	23
Bounces	10	34	18
Total	175	296	214
Contacts w/ No Reply	85.9%	87.2%	92.8%

Table 3: Email Responses—We received 685 email responses to our notifications, of which 14% were human replies.

Human Responses	ICS	IPv6	Ampl.
Positive Sentiments	17	35	19
Negative Sentiments	0	4	0
Neutral Sentiments	5	9	4
Request for Information	2	16	5
Taking Actions	12	17	15
False Positive Notification	0	12	2
Total	22	48	23

Table 4: Human Email Responses—We characterize the human email responses we received in reply to our notifications.

replies were in response to English notifications and expressed gratitude; none requested additional information.

We note that the level of feedback we received regarding DDoS notifications was commensurate with our other efforts, yet the patch response was minimal. This could indicate that operators struggle with actually resolving the issue after encountering and responding to our messages, or have become desensitized enough to DDoS issues to not take real action.

5.2 Anonymous Survey Responses

All of our notification messages contained a link to an anonymous seven question survey (Appendix A), to which we received 57 submissions. We summarize the results in Table 5.

Interestingly, 46% of respondents indicated that they were aware of the vulnerability prior to notification, and 16% indicated that they had previously attempted to resolve the problem. This contrasts with the survey results in the Heartbleed study [9], where all 17 respondents indicated they were aware of the Heartbleed vulnerability and had previously attempted to resolve the problem. The widespread media attention regarding the Heartbleed bug may account for this discrepancy, highlighting the differences in the nature of various vulnerabilities.

For DDoS amplifiers and ICS vulnerabilities, the ma-

Survey Responses	ICS	IPv6	Ampl.
Aware of Issue	2/4	20/45	4/8
Taken Prior Actions	1/4	5/43	3/8
Now Taking Action	4/4	24/43	6/8
Acceptable to Detect	3/4	35/45	7/8
Acceptable to Notify	2/4	34/45	7/8
Would want Future Notifications	2/4	30/43	7/8
Correct Contact	1/3	37/43	6/8
Total	4	45	8

Table 5: Survey Responses—We included a link to a short, anonymous survey in all of our notifications. We find that most respondents (54%) weren’t aware of the vulnerabilities, but found our scanning and notifications acceptable (over 75%). Further, 62% of respondents stated they were taking corrective actions and 71% of respondents requested future notifications.

jority of respondents expressed that they were now taking corrective action (75% for DDoS amplifiers, 100% for ICS). For IPv6, only 56% of respondents indicated they would fix the problem. Given the nature of the IPv6 notification, it is likely that some of the misaligned policies were intentional.

Over 80% of respondents indicated that we reached out to the correct contact, who found scanning and notifications acceptable and requested future vulnerability notifications. However, this is a population with whom we successfully established communication. The accuracy of the other contacts from whom we did not hear back could be lower.

Our survey also allowed respondents to enter free form comments. We received 17 IPv6 comments, 4 DDoS amplifier comments, and 1 ICS comment. Of the IPv6 respondents, 5 thanked us, 7 discussed how the misalignment could be intentional or that our detection was incorrect, 3 equated our messages to spam, and 2 noted that they initially thought our translated messages were phishing messages because they expected English messages from an institution in the United States. For amplifiers, we received four comments: two thanking us and two informing us not to notify unless there is a real attack. Finally, there was only one ICS commenter, who suggested contacting vendors instead of network operators, but thanked us for our notification.

The feedback we received from these survey answers and the email responses indicates an overall positive reception of our notifications. While it may be that those who provided feedback are more opinionated, these results suggest that further discourse on notifications is needed within our community.

6 Discussion

Here we summarize the main results developed during our study, and the primary avenues for further work that these suggest.

Effective Vulnerability Notifications Our results indicate that vulnerability notifications can improve remediation behavior and the feedback we received from network operators was largely positive. We conclude that notifications are most effective when detailed messages are sent directly to WHOIS abuse contacts. These notifications were most effective in our experiments and resulted in an additional 11% of contacts addressing a vulnerability in response to our message.

On the one hand, this result provides clear guidance on how to best notify network operators. On the other hand, the majority of organizations did not patch their hosts despite our notifications. Even among those who patched at least one host, most did not fix all of their vulnerable hosts. In the case of networks hosting DDoS amplifiers, *no* form of notification generated benefits statistically significant over the control.

The failures to remediate could signal a number of problems, including:

1. failure to contact the proper parties who could best instigate remediation;
2. a need for better education about the significance of the vulnerability;
3. a need for better education about the remediation process;
4. administrative or logistical hurdles that proved too difficult for those parties to overcome;
5. or a cost-benefit analysis by those parties that concluded remediation was not worth the effort.

Illuminating the role that each of these considerations plays, and the best steps to then address them, remains for future work.

In addition, we found the effects of our notification campaigns to be short-lived: if recipients did not act within the first couple days, they were unlikely to ever do so. Repeat notifications did not further improve remediation levels.

Thus, while we have developed initial guidance for conducting effective notifications, there remain many unanswered questions as to how to best encourage operators to patch vulnerable hosts.

Improving Centralized Notification Mechanisms We observed that relying on national and regional CERT organizations for vulnerability notifications had either a modest effect (compared to our direct notifications) or no effect (indistinguishable from our unnotified controls).

While certain national CERTs evinced improved levels of remediation, others either did not act upon the information we reported, or if they did so, recipients ignored their messages. Thus, the community should consider more effective mechanisms for facilitating centralized reporting, either within the existing CERT system, or using some separate organizational structure. This need is quite salient because the burden of locating and messaging thousands of individual contacts is high enough that many researchers will find it too burdensome to conduct notifications themselves.

Open Ethical Questions The process of notifying parties regarding security issues raises a number of ethical questions. The community has already discussed some of these in depth, as in the debates concerning “full disclosure.” Contacting individual sites suffering from vulnerabilities, likewise, raises questions regarding appropriate notification procedures.

For example, WHOIS abuse emails are a point-of-contact that multiple notification efforts have relied on [5, 9, 13, 14, 22, 24]. However, these contacts are technically designated for reports of abusive, malicious behavior (a point noted in the feedback we received as detailed in Section 5). While vulnerability reports have a somewhat similar flavor, they do not serve the same purpose. It behooves the security community to establish a standardized and reliable point-of-contact for communicating security issues.

Another question concerns whether the benefits of repeated notifications for the same vulnerability outweigh the costs imposed on recipients. Some may derive no benefits from the additional messages due to having no means to effectively remediate, yet must spend time ingesting the notifications. From our results, we observed that repeat notifications did not promote further patching, which argues against performing re-notifications.

More provocative, and related to the full-disclosure debate mentioned above, is the notion of *threatening* recipients with publicly revealing their vulnerabilities if unaddressed after a given amount of time. Likely, the research community would find this (quite) unpalatable in general; however, one can imagine specific situations where the community might conclude that spurring vital action justifies such a harsh step, just as some have concluded regarding full disclosure.

Future Abuse of Notifications In a future with widespread notifications, we would hope that security issues could be rectified more extensively and quickly. However, this would provide a new avenue for abuse, as attackers could potentially leverage the open communication channel to target network operators. As a simple example, a malicious actor could notify operators about a real security issue, and inform the operators to install

a malicious application to help hosts resolve the security gap. While existing techniques such as phishing detection and binary analysis can help limit these attacks, the problem domain likely will yield new challenges. It is important that the security community remain cognizant of these dangers as the state of security notifications evolves.

Effective Remediation Tools For contacts that do not remediate, our measurements cannot distinguish which of the underlying reasons sketched above came into play. However, while some operators may lack sufficient motivation to take action, it seems quite plausible that others wish to, but lack the technical capabilities, resources, or permissions to do. Accordingly, we see a need for investigation into the operational problems that operators encounter when considering or attempting remediation, as well as the development of effective and usable remediation tools that simplify the operators’ tasks. By reducing the effort and resources required to address a vulnerability, such tools could also increase the likelihood that an operator would take the steps to react to vulnerability reports. Ultimately, automated systems would be ideal, but these face significant challenges, such as heterogeneous platforms, potential abusive or malicious behavior, and inadvertent disruption of mission-critical systems.

7 Conclusion

We have undertaken an extensive study of notifying thousands of network operators of security issues present within their networks, with the goal of illuminating which fundamental aspects of notifications have the greatest impact on efficacy. Our study investigated vulnerabilities that span a range of protocols and considerations: exposure of industrial control systems; apparent firewall omissions for IPv6-based services; and exploitation of local systems in DDoS amplification attacks.

Through controlled multivariate experiments, we studied the impact of a number of variables: choice of party to contact (WHOIS abuse contacts versus national CERTs versus US-CERT), message verbosity, hosting a website linked to in the message, and translating the message into the notified party’s local language. We monitored the vulnerable systems for several weeks to determine their rate of remediation in response to changes to these variables.

We also assessed the outcome of the emailing process itself and characterized the sentiments and perspectives expressed in both the human replies and an optional anonymous survey that accompanied our notifications. The responses were largely positive, with 96% of human email responses expressing favorable or neutral sentiments.

Our findings indicate that notifications can have a significant positive effect on patching, with the best messaging regimen being directly notifying contacts with detailed information. An additional 11% of contacts addressed the security issue when notified in this fashion, compared to the control. However, we failed to prompt the majority of contacts to respond, and even when they did, remediation was often only partial. Repeat notifications did not further improve remediation. Given these positive yet unsatisfactory outcomes, we call on the security community to more deeply investigate notifications and establish standards and best practices that promote their effectiveness.

Acknowledgments

The authors thank L. Aaron Kaplan for insightful discussions regarding the CERT organizations and Philip Stark for providing statistical consultation. We similarly thank Jethro Beekman, Christian Kreibich, Kirill Levchenko, Philipp Moritz, Antonio Puglielli, and Matthias Vallentin for message translations. Additionally, we thank the reviewers and our shepherd Nicolas Christin for helpful feedback.

This work was supported in part by the National Science Foundation under contracts 1111672, 1111699, 1237264, 1237265, 1345254, 1409505, 1409758, 1518741, 1518888, 1518921, and 1619620. The first author is supported by a National Science Foundation Graduate Research Fellowship. The second author is supported by the Google Ph.D. Fellowship in Computer Security. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of any funding sponsor.

References

- [1] National Vulnerability Database. <https://nvd.nist.gov/>.
- [2] Conficker Working Group: Lessons Learned, 2011. http://www.confickerworkinggroup.org/wiki/uploads/Conficker_Working_Group_Lessons_Learned_17_June_2010_final.pdf.
- [3] CAIDA. Archipelago (Ark) Measurement Infrastructure. <http://www.caida.org/projects/ark/>.
- [4] CENTERS FOR DISEASE CONTROL AND PREVENTION. Patient Notification Toolkit. <http://www.cdc.gov/injectionsafety/pntoolkit/index.html>.
- [5] CETIN, O., JHAVERI, M. H., GANAN, C., EETEN, M., AND MOORE, T. Understanding the Role of Sender Reputation in Abuse Reporting and Cleanup. In *Workshop on the Economics of Information Security (WEIS)* (2015).
- [6] CZYZ, J., LUCKIE, M., ALLMAN, M., AND BAILEY, M. Don't Forget to Lock the Back Door! A Characterization of IPv6 Network Security Policy. In *Symposium on Network and Distributed System Security (NDSS)* (2016).
- [7] DITTRICH, D., BAILEY, M., AND DIETRICH, S. Towards Community Standards for Ethical Behavior in Computer Security Research. Tech. rep., 2009.
- [8] DURUMERIC, Z., ADRIAN, D., MIRIAN, A., BAILEY, M., AND HALDERMAN, J. A. A Search Engine Backed by Internet-Wide Scanning. In *ACM Conference on Computer and Communications Security (CCS)* (2015).
- [9] DURUMERIC, Z., LI, F., KASTEN, J., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., ADRIAN, D., PAXSON, V., BAILEY, M., AND HALDERMAN, J. A. The Matter of Heartbleed. In *ACM Internet Measurement Conference (IMC)* (2014).
- [10] DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *USENIX Security Symposium* (2013).
- [11] GRAHAM, R. Masscan: The Entire Internet in 3 Minutes. Errata Security Blog, 2013. <http://blog.erratasec.com/2013/09/masscan-entire-internet-in-3-minutes.html>.
- [12] HOFMEYR, S., MOORE, T., FORREST, S., EDWARDS, B., AND STELLE, G. Modeling Internet-Scale Policies for Cleaning up Malware. In *Economics of Information Security and Privacy III* (2013).
- [13] KUHRER, M., HUPPERICH, T., ROSSOW, C., AND HOLZ, T. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *USENIX Security Symposium* (2014).
- [14] LI, F., HO, G., KUAN, E., NIU, Y., BALLARD, L., THOMAS, K., BURSZTEIN, E., AND PAXSON, V. Remedyng Web Hijacking: Notification Effectiveness and Webmaster Comprehension. In *World Wide Web Conference (WWW)* (2016).
- [15] LUCKIE, M. Scamper: A Scalable and Extensible Packet Prober for Active Measurement of the Internet. In *ACM Internet Measurement Conference (IMC)* (2010).
- [16] MAXMIND, LLC. Geoip2 database.
- [17] MIRIAN, A., MA, Z., ADRIAN, D., TISCHER, M., CHUENCHUIJT, T., MASON, J., YARDLEY, T., BERTHIER, R., DURUMERIC, Z., HALDERMAN, J. A., AND BAILEY, M. An Internet-Wide View of Publicly Accessible SCADA Devices. Unpublished Manuscript.
- [18] MOORE, T., AND CLAYTON, R. Ethical Dilemmas in Take-down Research. In *International Conference on Financial Cryptography and Data Security (FC)* (2011).
- [19] NATIONAL CENTER FOR BIOTECHNOLOGY INFORMATION. Clinical Effectiveness of Partner Notification. <http://www.ncbi.nlm.nih.gov/books/NBK261439/>.
- [20] RAPID7. DNS Records (ANY) Dataset, 2015. <https://scans.io/study/sonar.fdns>.
- [21] ROSSOW, C. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Symposium on Network and Distributed System Security (NDSS)* (2014).
- [22] STOCK, B., PELLEGRINO, G., ROSSOW, C., JOHNS, M., AND BACKES, M. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *USENIX Security Symposium* (2016).
- [23] STONE-GROSS, B., CAVALLARO, L., GILBERT, B., SZYDŁOWSKI, M., KEMMERER, R., KRUEGEL, C., AND VIGNA, G. Your Botnet Is My Botnet: Analysis of a Botnet Takeover. In *ACM Conference on Computer and Communications Security (CCS)* (2009).
- [24] VASEK, M., AND MOORE, T. Do Malware Reports Expedite Cleanup? An Experimental Study. In *USENIX Workshop on Cyber Security Experimentation and Test (CSET)* (2012).
- [25] YEGNESWARAN, V., BARFORD, P., AND PAXSON, V. Using Honeynets for Internet Situational Awareness. In *Hot Topics in Networks (HotNets)* (2005).

A Anonymous and Optional Security Notifications Survey

Help us better understand the factors surrounding security notifications by providing anonymous feedback in this survey. Each question is optional, so answer the ones you feel comfortable answering. Thank you!

1. Was your organization aware of the security issue prior to our notification?
2. Did your organization take prior actions to resolve the security issue before our notification?
3. Is your organization planning on resolving the security issue?
4. Do you feel it was acceptable for us to detect the security issue?
5. Do you feel it was acceptable for us to notify your organization?
6. Would your organization want to receive similar security vulnerability/misconfiguration notifications in the future?
7. Did we notify the correct contact?

B IPv6 Notification: Terse with Link

Subject: [RAND#] Potentially Misconfigured IPv6 Port Security Policies

Body: Computer scientists at the University of Michigan, the University of Illinois Urbana-Champaign, and the University of California Berkeley have been conducting Internet-wide scans to detect IPv4/IPv6 dual-stack hosts that allow access to services via IPv6, but not IPv4. This likely indicates a firewall misconfiguration and could be a security vulnerability if the services should not be publicly accessible. We have attached a list of hosts that are potentially vulnerable on your network.

[LINK: More information is available at [https://security-notifications.cs.berkeley.edu/\[RAND#/\]ipv6.html](https://security-notifications.cs.berkeley.edu/[RAND#/]ipv6.html).]

Thank you,

Berkeley Security Notifications Team

Help us improve notifications with anonymous feedback at: <https://www.surveymonkey.com/r/Q2HLJ5D>

C IPv6 Notification:Verbose

Subject: [RAND#] Potentially Misconfigured IPv6 Port Security Policies

Body: During a recent study on the network security policies of IPv4/IPv6 dual-stack hosts, computer scientists at the University of Michigan, the University of Illinois Urbana-Champaign, and the University of California Berkeley have been conducting Internet-wide scans to detect IPv4/IPv6 dual-stack hosts that allow access to services via IPv6, but not IPv4. This likely indicates a firewall misconfiguration and could be a security vulnerability if the services should not be publicly accessible. We have attached a list of hosts that are potentially vulnerable on your network (as determined by WHOIS information).

For each dual-stack host, we test whether popular services (e.g., SSH, Telnet, and NTP) are accessible via IPv4 and/or IPv6 using a standard protocol handshake. For ICMP this is an echo request, for TCP it is a SYN segment, and for UDP this is an application-specific request (e.g., DNS A query for ‘www.google.com’ or an NTP version query). We do not exploit any vulnerabilities, attempt to login, or access any non-public information.

The protocols we scanned are popular targets for attack and/or can be used to launch DDoS attacks when left publicly available to the Internet. We suspect they are misconfigured and are notifying you because hosts rarely offer services on IPv6 that are not offered on IPv4, and we believe these services may have been left exposed accidentally. This is a common occurrence when administrators forget to configure IPv6 firewall policies along with IPv4 policies.

If these IPv6-only accessible services should not be accessible to the public Internet, they can be restricted by updating your firewall or by disabling or removing the services. If none of your systems use IPv6, you can also disable IPv6 on your system. Make sure your changes are persistent and will not be undone by a system reboot.

More information is available at [https://security-notifications.cs.berkeley.edu/\[RAND#/\]ipv6.html](https://security-notifications.cs.berkeley.edu/[RAND#/]ipv6.html).

Thank you,

Berkeley Security Notifications Team

Help us improve notifications with anonymous feedback at: <https://www.surveymonkey.com/r/Q2HLJ5D>

D ICS Notification: Terse with Link

Subject: [RAND#] Vulnerable SCADA Devices

Body: Computer scientists at the University of Michigan and the University of California Berkeley have been conducting Internet-wide scans to detect publicly accessible industrial control (SCADA) devices. These devices frequently have no built-in security and their public exposure may place physical equipment at risk for attack. We have attached a list of SCADA devices on your network that are publicly accessible.

[LINK: More information is available at [https://security-notifications.cs.berkeley.edu/\[RAND#\]/ics.html](https://security-notifications.cs.berkeley.edu/[RAND#]/ics.html).]

Thank you,

Berkeley Security Notifications Team

Help us improve notifications with anonymous feedback at: <https://www.surveymonkey.com/r/ZC7BVW5>

nal, segmented network, or otherwise protected by a firewall that limits who can interact with these hosts. Make sure your changes are persistent and will not be undone by a system reboot.

More information is available at [https://security-notifications.cs.berkeley.edu/\[RAND#\]/ics.html](https://security-notifications.cs.berkeley.edu/[RAND#]/ics.html).

Thank you,

Berkeley Security Notifications Team

Help us improve notifications with anonymous feedback at: <https://www.surveymonkey.com/r/ZC7BVW5>

E ICS Notification: Verbose

Subject: [RAND#] Vulnerable SCADA Devices

Body: During a recent study on the public exposure of industrial control systems, computer scientists at the University of Michigan and the University of California Berkeley have been conducting Internet-wide scans to detect publicly accessible industrial control (SCADA) devices. These devices frequently have no built-in security and their public exposure may place physical equipment at risk for attack. We have attached a list of SCADA devices on your network (as determined by WHOIS information) that are publicly accessible.

We scan for potentially vulnerable SCADA systems by scanning the full IPv4 address space and attempting protocol discovery handshakes (e.g., Modbus device ID query). We do not exploit any vulnerabilities or change any device state.

SCADA protocols including Modbus, S7, Bacnet, Tridium Fox, and DNP3 allow remote control and monitoring of physical infrastructure and equipment over IP. Unfortunately, these protocols lack critical security features, such as basic authentication and encryption, or have known security vulnerabilities. If left publicly accessible on the Internet, these protocols can be the target of attackers looking to monitor or damage physical equipment, such as power control, process automation, and HVAC control systems.

SCADA services are not designed to be publicly accessible on the Internet and should be maintained on an inter-

F DDoS Amplification Notification: Terse with Link

Subject: [RAND#] Vulnerable DDoS Amplifiers

Body: Computer scientists at George Mason University and the University of California Berkeley have been detecting open and misconfigured services that serve as amplifiers for distributed denial-of-service (DDoS) attacks. Attackers abuse these amplifiers to launch powerful DDoS attacks while hiding the true attack source. We have attached a list of hosts that are potentially vulnerable on your network.

[LINK: More information is available at [https://security-notifications.cs.berkeley.edu/\[RAND#\]/amplifiers.html](https://security-notifications.cs.berkeley.edu/[RAND#]/amplifiers.html).]

Thank you,

Berkeley Security Notifications Team

Help us improve notifications with anonymous feedback at: <https://www.surveymonkey.com/r/Y99J8K8>

G DDoS Amplification Notification: Verbose

Subject: [RAND#] Vulnerable DDoS Amplifiers

Body: During a recent study on distributed denial-of-service (DDoS) attacks, computer scientists at George Mason University and the University of California Berkeley have been conducting Internet-wide scans for open and misconfigured services that serve as amplifiers for DDoS attacks. Attackers abuse these amplifiers to launch powerful DDoS attacks while hiding the true at-

tack source. We have attached a list of hosts that are potentially vulnerable on your network (as determined by WHOIS information).

We detect amplifiers by monitoring hosts involved in recent DDoS attacks and checking whether these hosts support the features used for launching an attack (e.g., NTP monlist or recursive DNS resolution). We do not exploit any vulnerabilities or attempt to access any non-public data on these servers.

DDoS attacks are often conducted by directing an overwhelming amount of network traffic towards a target system, making it unresponsive. Amplifiers are services that send large amounts of data in response to small requests. Attackers leverage these in DDoS attacks by spoofing traffic to the amplifier, forging it to look as if it came from the attacker's target. Amplifiers then respond to the target with a large response that overwhelms the target. Publicly accessible amplifiers are constantly abused by attackers to conduct the DDoS attacks for them while hiding the tracks of the real attacker.

These amplifiers can be avoided by disabling the application or updating your firewall to block the application port or restrict the IP addresses that can access it. More specifically, Chargen should be closed as it is rarely useful and is inherently an amplifier. If left open, DNS should be configured to restrict who can make recursive requests, and NTP should be configured to disable the monlist functionality. Make sure your changes are persistent and will not be undone by a system reboot.

More information is available at [https://security-notifications.cs.berkeley.edu/\[RAND#\]/amplifiers.html](https://security-notifications.cs.berkeley.edu/[RAND#]/amplifiers.html).

Thank you,

Berkeley Security Notifications Team

Help us improve notifications with anonymous feedback
at: <https://www.surveymonkey.com/r/Y99J8K8>

Mirror: Enabling Proofs of Data Replication and Retrievability in the Cloud

Frederik Armknecht

University of Mannheim, Germany

armknecht@uni-mannheim.de

Jens-Matthias Bohli

NEC Laboratories Europe, Germany

Hochschule Mannheim, Germany

jens.bohli@neclab.eu

Ludovic Barman

NEC Laboratories Europe, Germany

ludovic.barman@neclab.eu

Ghassan O. Karame

NEC Laboratories Europe, Germany

ghassan.karame@neclab.eu

Abstract

Proofs of Retrievability (POR) and Data Possession (PDP) are cryptographic protocols that enable a cloud provider to prove that data is correctly stored in the cloud. PDP have been recently extended to enable users to check in a single protocol that additional file replicas are stored as well. To conduct multi-replica PDP, users are however required to process, construct, and upload their data replicas by themselves. This incurs additional bandwidth overhead on both the service provider and the user and also poses new security risks for the provider. Namely, since uploaded files are typically encrypted, the provider cannot recognize if the uploaded content are indeed replicas. This limits the business models available to the provider, since e.g., reduced costs for storing replicas can be abused by users who upload different files—while claiming that they are replicas.

In this paper, we address this problem and propose a novel solution for proving data replication and retrievability in the cloud, **Mirror**, which allows to shift the burden of constructing replicas to the cloud provider itself—thus conforming with the current cloud model. We show that **Mirror** is secure against malicious users and a rational cloud provider. Finally, we implement a prototype based on **Mirror**, and evaluate its performance in a realistic cloud setting. Our evaluation results show that our proposal incurs tolerable overhead on the users and the cloud provider.

1 Introduction

The cloud promises a cost-effective alternative for small and medium enterprises to downscale/upscale their services without the need for huge upfront investments, e.g., to ensure high service availability.

Currently, most cloud storage services guarantee

service and data availability [4, 6] in their Service Level Agreements (SLAs). Availability is typically ensured by means of full replication [4, 23]. Replicas are stored onto different servers—thus ensuring data availability in spite of server failure. Storage services such as Amazon S3 and Google FS provide such resiliency against a maximum of two concurrent failures [30]; here, users are typically charged according to the required redundancy level [4].

Nevertheless, none of today’s cloud providers accept any liability for data loss in their SLAs. This makes users reluctant—and rightly so—when using cloud services due to concerns with respect to the integrity of their outsourced data [2, 7, 10]. These concerns have been recently fueled by a number of data loss incidents within large cloud service providers [5, 10]. For instance, Google recently admitted that a small fraction of their customers’ data was permanently lost due to lightning strikes which caused temporary electricity outages [10].

To remedy this, the literature features a number of solutions that enable users to remotely verify the availability and integrity of stored data [11, 15, 16, 25, 34]. Examples include Proofs of Retrievability (POR) [25, 34] which provide clients with the assurance that their data is available in its entirety, and Proofs of Data Possession (PDP) [12] which enable a client to verify that its stored data has not undergone any modifications. PDP schemes have been recently extended to verify the replication of files [18, 22, 30]. These extensions can provide guarantees for the users that the storage provider is replicating their data as agreed in the SLA, and that they are indeed getting the value for their money.

Notice, however, that existing solutions require the users themselves to create replicas of their files, appropriately pre-process the replicas (i.e., to create authentication tags for PDP), and finally upload all processed replicas in the cloud. Clearly, this in-

Table 1: Bandwidth cost in different regions as provided by CloudFlare [3]. “% Peered” refers to the percentage of traffic exchanged for free with other providers.

Region	% Peered	Effective price/Mbps/Month
Europe	50%	\$5
North America	20%	\$8
Asia	55%	\$32
Latin America	60%	\$32
Australia	50%	\$100

curs significant burden on the users. Moreover, this consumes considerable bandwidth from the provider, that might have to scale up its bandwidth to accommodate for such large upload requests. For example, in order to store a 10 GB file together with three replicas, users have to process and upload at least 40 GB of content. Recall that the provider’s bandwidth is a scarce resource; most providers, such as AWS and CloudFlare, currently buy bandwidth from a number of so-called Tier 1 providers to ensure global connectivity to their datacenters [3]. For example, CloudFlare pays for maximum utilization (i.e., maximum number of Mbps) used per month. This process is costly (cf. Table 3) and is considerably more expensive than acquiring storage and computing resources [24].

Besides consuming the provider’s bandwidth resources, this also limits the business models available to the provider, since e.g., reduced costs for storing replicas can be offered in the case where the replication process does not consume considerable bandwidth resources from the provider (e.g., when the replication is locally performed by the provider). Alternatively, providers can offer reduced costs by offering infrequent/limited access to stored replicas, etc. Amazon S3, for example, charges its users almost 25% of the underlying storage costs for additional replication [1,9]. Users therefore have considerable incentives to abuse this service, and to store their data at reduced costs as if they were replicas. Since the outsourced data is usually encrypted, the provider cannot recognize if the uploaded contents are indeed replicas.

In this paper, we address this problem, and propose a novel solution for proving data replication and retrievability in the cloud, **Mirror**, which goes beyond existing multi-replica PDP solutions and enables users to efficiently verify the retrievability of all their replicas without requiring them to replicate data by themselves. Notably, in **Mirror**, users need to process/upload their original files only once irrespective of the replication undergone by their data; here, conforming with the current cloud model [4],

the cloud provider appropriately constructs the replicas given the original user files. Nevertheless, **Mirror** allows users to efficiently verify the retrievability of all data replicas—including those constructed by the service provider.

To this end, **Mirror** leverages cryptographic puzzles to impose significant resource constraints—and thus an economic disincentive—on a cloud provider which creates the replicas on demand, i.e., whenever the client initiates the verification protocol. By doing so, **Mirror** incentivizes a rational cloud provider to correctly store and replicate the clients’ data—otherwise the provider risks detection with significant probability.

In summary, we make the following contributions in this work:

- We propose a novel formal model and a security model for proofs of replication and retrievability. Our proposed model, **PoR**², extends the POR model outlined in [34] and addresses security risks that have not been covered so far in existing multi-replica PDP models.
- We describe a concrete **PoR**² scheme, dubbed **Mirror** that is secure in our enhanced security model. **Mirror** leverages a tunable replication scheme based on the combination of Linear Feedback Shift Registers (LFSRs) with the RSA-based puzzle by Rivest [33]. By doing so, **Mirror** shifts the burden of constructing replicas to the cloud provider itself and is therefore likely to be appreciated by cloud providers since it allows them to trade their expensive bandwidth resources with relatively cheaper computing resources.
- We implement and evaluate a prototype based on **Mirror** in a realistic cloud setting, and we show that our proposal incurs tolerable overhead on both the users and the cloud provider when compared to existing multi-replica PDP schemes.

The remainder of this paper is organized as follows. In Section 2, we introduce a novel model for proofs of replication and retrievability. In Section 3, we propose **Mirror**, an efficient instantiation of our proposed model, and analyze its security in Section 4. In Section 5, we evaluate a prototype implementation of **Mirror** in realistic cloud settings and compare its performance to the multi-replica PDP scheme of [18]. In Section 6, we overview related work in the area, and we conclude the paper in Section 7.

2 PoR²: Proofs of Replication and Retrievability

In this section, we introduce a formal model for proofs of replication and retrievability, PoR².

2.1 System Model

We consider a setting where a user \mathcal{U} outsources a file D to a service provider \mathcal{S} who agrees to the following two conditions:

1. Store the file D in its entirety.
2. Additionally store r replicas of D in their entirety.

A PoR² protocol aims to ensure to the user that both conditions are kept without the need for users to download the files and the replicas. Hence, our model comprises a further party: the verifier \mathcal{V} who runs the PoR² scheme to validate that indeed the data and sufficient copies are stored by \mathcal{S} . In a privately-verifiable scheme, the user and the verifier consist of the same entity; these roles might be however different in publicly-verifiable schemes.

As one can see, Condition 1 indirectly implies that a PoR² scheme needs to comprise a PDP or POR scheme. Consequently, similar to the POR model, a PoR² involves a process for outsourcing the original data, referred to as **Store**, and an interactive verification protocol **Verify**.

However, Condition 2 goes beyond common POR/PDP requirements. Hence, one needs an additional (possibly interactive) process denoted by **Replicate** for generating the replicas and a second process, dubbed **CheckReplica**, which checks the correctness of the replicas (in case the replicas were created by the user). Moreover, the interactive verification protocol **Verify** needs to be extended such that it verifies the storage of the original file *and* the copies computed by the service provider. In what follows, we give a formal specification of the procedures and protocols involved in a PoR² scheme. Our model adapts and extends the original POR model introduced in [25, 34]. In Section 2.2, we summarize the relation between PoR² and previous POR models.

The Store Procedure: This randomized procedure is executed by the user once at the start. **Store** takes as input the security parameter κ and the file \tilde{D} to be outsourced, and produces a file tag τ that is required to run the verification procedure. Depending on whether the scheme is private or public verifiable, the verification tag needs to be kept secret or can be made public. The output of **Store** comprises the file D that the service provider should store. D may

be different from \tilde{D} , e.g., contain some additional information, but \tilde{D} should be efficiently recoverable from D . Finally, **Store** outputs public parameters Π which allow the generation of r replicas $D^{(i)}$ of the outsourced file. We assume that the number of copies is (implicitly) given in the copy parameters Π , possibly being specified in the SLA before. Summing up, the formal specification of **Store** is:

$$(D, \tau, \Pi) \leftarrow \text{Store}(\kappa, \tilde{D})$$

The Replicate Procedure: The **Replicate** procedure is a protocol executed between the verifier (who holds the verification tag τ) and the service provider to generate replicas of the original file. To this end, **Replicate** takes as inputs the copy parameters Π and the outsourced file D , and outputs the r copies $D^{(1)}, \dots, D^{(r)}$. In addition, the provider gets a *copy tag* τ^* which allows him to validate the correctness of the copies. Formally, we have:

$$\begin{aligned} \text{Replicate} : & \quad [\mathcal{V} : \tau, \Pi; \mathcal{S} : D, \Pi] \\ & \rightarrow [\mathcal{V} : \tau; \mathcal{S} : D^{(1)}, \dots, D^{(r)}, \tau^*] \end{aligned}$$

Recall that the verifier \mathcal{V} refers to the party that holds the verification tag and may not necessarily be a third party. This captures (i) the case where the user creates the copies on his own at the beginning (as discussed in [18]), and (ii) the case where this replication process is completely outsourced to the service provider (or even to a third party).

Observe that the output for the verifier includes again the verification tag. This is the case since we want to capture situations where the verification tag can be changed, depending on the protocol flow of **Replicate**. To keep the description simple, we denote both values (the verification tag as output of the **Store** procedure and the potentially updated verification tag after running **Replicate**) by τ .

The CheckReplica Procedure: The purpose of the **CheckReplica** procedure, which is used by the service provider, is to validate that the replicas have been correctly *generated*, i.e., are indeed copies of the original file. Notice that **CheckReplica** is mandatory for a comprehensive model but is not necessary in the case where the service provider replicates the data itself (in this case the service provider can ensure that the replication process is done correctly).

CheckReplica is executed between the verifier and the service provider. The verifier \mathcal{V} takes as input the copy parameters Π and verification tag τ , being his output of the **Replicate** procedure (see above), while the service provider \mathcal{S} takes as input the uploaded file D , a possible replica D^* (together with a replica

index $i \in \{1, \dots, r\}$), the copy parameters Π , and the copy tag τ^* . CheckReplica then outputs a binary decision expressing whether the service provider \mathcal{S} believes that D^* is a correct i -th replica of D according to the Replicate procedure and the copy parameters Π .

$\text{CheckReplica}: [\mathcal{V} : \tau, \Pi; \mathcal{S} : \tau^*, \Pi, D, D^*, i] \rightarrow [\mathcal{S} : \text{dec}]$

The Verify Protocol: A verifier \mathcal{V} , i.e., the user if the scheme is privately verifiable and possibly a third party if the scheme is publicly verifiable, and the provider \mathcal{S} execute an interactive protocol to convince the verifier that both the outsourced D and the r replicas $D^{(1)}, \dots, D^{(r)}$ are correctly stored. The input of \mathcal{V} is the tag τ given by Store and the copy parameters Π , while the input of the provider \mathcal{S} is the file D outsourced by the user and the r replicas generated by the Replicate procedure. The output $\text{dec} \in \{\text{accept}, \text{reject}\}$ of the verifier expresses his decision, i.e., whether he accepts or rejects. It holds that:

$\text{Verify}: [\mathcal{V} : \tau, \Pi; \mathcal{S} : D, D^{(1)}, \dots, D^{(r)}] \longrightarrow [\mathcal{V} : \text{dec}]$

Note that Verify and CheckReplica aim for completely different goals. The CheckReplica procedure allows the service provider \mathcal{S} to check if the replicas have been *correctly generated* and hence protects against a malicious customer who misuses the replicas for storing additional data at lower costs. On the other side, the Verify procedure enables a client or verifier \mathcal{V} to validate that the file and all copies are indeed *stored in their entirety* to provide security against a rational service provider. For instance, CheckReplica can be omitted if the replicas have been generated by the service provider directly while Verify would still be required.

2.2 Relation to Previous Models

Notice that the introduced PoR^2 model covers and extends both proofs of retrievability and proofs of multiple replicas. For example in case that no replicas are created at all, i.e., neither the Replicate nor the CheckReplica procedures are used, the scheme reduces to a standard POR according to the model given in [34]. Observe that in such cases storage allocation is a direct consequence of the incompressibility of the file. Moreover, the multi-replica schemes presented so far (see Section 6 for an overview) can be seen as PoR^2 schemes where the correct replication requirement is simply ignored. In fact, we argue that if existing multi-replica schemes are coupled with a proof that the replicas are honestly generated by the user, then the resulting scheme would be a secure PoR^2 scheme.

2.3 Attacker Model

Similar to existing work in the area [35, 36], we adapt the concept of the *rational attacker model*. Here, rational means that if the provider cannot save any costs by misbehaving, then he is likely to simply behave honestly. In our particular case, the advantage of the adversary clearly depends on the relation between storage costs and other resources (such as computation), and on the availability of these resources to the adversary. In the sequel, we capture such a rational adversary by restricting him to a bounded number of concurrent threads of execution. Given that the provisioning of computational resources would incur additional costs, our restriction is justified by the fact that a rational adversary would only invest in additional computing resources if such a strategy would result in lower total costs (including the underlying costs of storage).

Likewise, we assume that users are interested to misuse the replicas for storing more data than has been agreed upon. Recall that since replicas are typically charged less than original files [1, 9], a rational user may try to encode additional information or other files into the replicas.

2.4 Security Goals and Correctness

In this section, we formalize the security goals of a PoR^2 scheme and define the correctness requirements. Note that we do not consider confidentiality of the file \tilde{D} , since we assume that the user encrypts the file prior to the start the PoR^2 protocol. We start by defining three security notions that a PoR^2 scheme must guarantee:

Extractability: The user can recover the uploaded file D .

Storage Allocation: Provider uses *at least* as much storage as required to store the file and all replicas.

Correct Replication: The files $D^{(i)}$ are *correct* replicas of D .

The extractability notion protects the user against a malicious service provider who does not store the whole file. Similarly, the storage allocation notion aims to protect a user against a service provider who does not commit enough storage to store all replicas. Clearly, the first two conditions together imply that a rational provider \mathcal{S} indeed stores D and the replicas $D^{(1)}, \dots, D^{(r)}$ and therefore fulfills his part of providing redundancy to protect the data. In contrast to the two previous notions, correct replication aims to

protect the service provider against a malicious user who tries to encode additional data in the replicas. This is an important property, which is not satisfied by existing multi-replica PDP models, but which should cater to any practical deployment of PoR². In Section 3, we propose an instantiation of PoR² which allows the provider to run `Replicate` by itself—thus inherently satisfying this property. In the following paragraphs, we provide a formal description of the above defined notions.

Extractability. Extractability guarantees that an honest user is able to recover the data \tilde{D} . Adopting [25, 34], this is formalized as follows. If a service provider is able to convince a honest user with significant probability during the `Verify` procedure, then there exists an *extractor algorithm* that can interact with the service provider and extract the file. This is captured by a hypothetical game between an adversary and an environment where the latter simulates all honest users and an honest verifier. The adversary is allowed to request the environment to create new honest users (including respective public and private keys), to let them store chosen files, and to run the `Verify` and `Replicate` procedures. At the end, the adversary chooses a user \mathcal{U} with the corresponding outsourced file D and outputs a service provider \mathcal{S} who can execute the `Verify` protocol with \mathcal{U} with respect to the chosen file D . We say that a service provider is ϵ -admissible if the probability that the verifier does not abort is at least ϵ .

Definition 1 (Extractability) *We say that a PoR² scheme is ϵ -extractable if there exists an extraction algorithm such that for any PPT algorithm who plays the aforementioned game and outputs an ϵ -admissible service provider \mathcal{S} , the extraction algorithm recovers D with overwhelming probability.*

In addition, we say that correctness is provided with respect to the extractability if the following holds. If all parties are honest, i.e., the user, the verifier, and the provider, then the verifier accepts the output of the `Verify` protocol with probability 1. This should hold for any file $\tilde{D} \in \{0, 1\}^$.*

Storage Allocation. Let ST denote the storage of the service provider that has been allocated for storing the file D and the replicas $D^{(1)}, \dots, D^{(r)}$. We compute the storage allocation by the provider, ρ , as follows:

$$\rho := \frac{|ST|}{|D| + |D^{(1)}| + \dots + |D^{(r)}|} \quad (1)$$

Here, we consider the generic case where the sizes of the replicas can be different (e.g., due to different

metadata). Moreover, we assume that neither the file nor the replicas can be (further) compressed, e.g., because these have been encrypted first. Since the service provider aims to save storage, it holds in general that $0 \leq \rho \leq 1$. Storage allocation ensures that $\rho \geq \delta$ for a threshold $0 \leq \delta \leq 1$ chosen by the user.

Definition 2 (Binding) *We say that a PoR² scheme is (δ, ϵ) -binding if for any rational attacker who plays the aforementioned game, and outputs an ϵ -admissible service provider \mathcal{S} who invests only a fraction $\rho < \delta$ of memory, it holds that the verifier accepts only with negligible probability (in the security parameter).*

We say that the scheme is even strongly (δ, ϵ) -binding if it holds for any PPT attacker, i.e., also for non-rational attackers.

We stress that the distinction between binding and strongly binding is necessary in a comprehensive model. For instance, for schemes where the replicas are generated locally by the service provider \mathcal{S} himself, i.e., to save bandwidth, the strongly binding property is impossible to achieve for $\delta > |ST|/|D|$. The reason is that a non-rational service provider could always store D only and run the `Replicate` procedure over and over again when needed. On the other hand, if the user is generating and uploading the replicas, strong binding could be achieved when replicas are different encryptions of the original file, e.g., as done in [18]. In Fortress, we aim to outsource the replica generation to the service provider to save bandwidth and hence only aim for the binding property.

Correct Replication. Correct replication means essentially that both, `Replicate` and `CheckReplica`, are sound and correct. We detail this below.

We say that `Replicate` is sound if in the case where the user is involved in the replica generation, the service provider can get assurance that the additionally uploaded data represents indeed correctly built replicas that do not encode, for example, some additional data. That is, `Replicate` must not be able to encode a significant amount of additional data in the replicas. This is formally covered by the requirement that inputs of the verifier to the replicate procedure `Replicate`, namely the verification tag τ and the copy parameters Π , have a size that is independent of the file size.

On the other hand, we say that `Replicate` is correct if replicas represent indeed copies of the uploaded file D . This is formally captured by requiring that

D can be efficiently recovered from any replica $D^{(k)}$. More precisely, we say that **Replicate** is correct if there exists an efficient algorithm which given τ , Π , and any replica $D^{(k)}$ outputs D .

With respect to **CheckReplica**, we require that \mathcal{S} only accepts replicas which are valid output of **Replicate**. Let D and Π be the output of the **Store** procedure. Let \mathcal{E} be the event that τ^* and $D^{(1)}, \dots, D^{(r)}$ are the output of a **Replicate** run. Let dec be the decision of the service provider at the end of the **CheckReplica** protocol. We say that the scheme is ϵ^* -correctly building replicas if:

$$\forall i \in \{1, \dots, r\} : \Pr[\text{dec} = \text{Accept} | \mathcal{E}] = 1, \\ \max_{i \in \{1, \dots, r\}} \{\Pr[\text{dec} = \text{Accept} | \neg \mathcal{E}]\} \leq \epsilon^*.$$

Observe that the first and second condition express the correctness and soundness of **CheckReplica**, respectively.

3 Mirror: An Efficient PoR² Instantiation

3.1 Overview

The goal of **Mirror** is to provide a verifiable replication mechanism for storage providers. Note that straightforward approaches to construct PoR² would either be communication-expensive or would be insecure in the presence of a rational cloud provider.

For instance, the user could create and upload the required t replicas of his files, similar to [18]. Obviously, this alternative incurs considerable bandwidth overhead on the providers and users can abuse the replicas to outsource several, different files in encrypted form. An alternative solution would be to enable the cloud provider to create the replicas (and their tags) on his own given the original files. This would significantly reduce the provider's bandwidth consumption incurred in existing multi-replica schemes at the expense of investing additional (cheaper) computing resources [24]. This alternative might be, however, insecure since it gives considerable advantage for the provider to misbehave, e.g., store only one single replica and construct the replicas on the fly when needed.

To thwart the generic attacks described above, **Mirror** ensures that a malicious cloud provider can only reply correctly within the verification protocol by investing a minimum amount of resources, i.e., memory and/or time. However, to ensure the binding property (Definition 2), i.e., that the provider invests memory and not time, **Mirror** allows to scale the computational effort that a dishonest provider would

have to invest without increasing the memory effort of an honest provider. This allows to adjust the computational effort of a dishonest provider such that the costs of storing the replicas is cheaper than the costs of computing the response to the challenges on the fly—giving an economic incentive to a rational provider to behave honestly.

This is achieved in **Mirror** through the use of a tunable puzzle-based replication scheme. Namely, in **Mirror**, the user has to outsource only his original files and compact puzzles to the cloud provider; the solution of these puzzles will be then combined with the original file in order to construct the r required replicas. Puzzles are constructed such that (i) they require noticeable time to be solved by the cloud provider while the user is significantly more efficient by exploiting a trapdoor, (ii) storing their solution incurs storage costs that are at least as large as the required storage for replicas, (iii) their difficulty can be easily adjusted by the creator to cater for variable strengths (and different cost metrics), and (iv) they can be efficiently combined with the original file blocks in order to create r correct replicas of the file preserving the homomorphic properties needed for compact proofs¹.

To this end, **Mirror** combines the use of the RSA puzzle of Rivest [33] and Linear Feedback Shift Registers (LFSR) (cf. Section 3.3). A crucial aspect here is that the user creates two LFSRs: a short one which is kept secret, and a longer public LFSR. The service provider is only given the public LFSR to generate the exponent values. As we show later, this allows for high degrees of freedom with respect to security and performance of **Mirror**. In the following, we first explain the deployed main building blocks and give afterwards the full protocol specification.

3.2 Building Blocks

RSA-based Puzzles: **Mirror** ties each sector with a cryptographic puzzle that is inspired by the RSA puzzle of Rivest [33]. In a nutshell, the puzzle requires the repeated exponentiation of given values $X^a \bmod N$ where $N = p \cdot q$ is publicly known RSA modulus and a, p, q remain secret. Without knowing these secrets, this requires to perform modular exponentiation. Modular exponentiation is an inherently sequential process [33]. The running time of the fastest known algorithm for modular exponentiation is linear in the size of the exponent. Although the provider might try to parallelize the computation of

¹This condition restricts our choice of puzzles since e.g., hash-based puzzles cannot be efficiently combined with the authentication tag of each data block/sector.

the puzzle, the parallelization advantage is expected to be negligible [17, 26, 28, 33]. On the other hand, the computation can be efficiently verified by the puzzle generator through the trapdoor offered by Euler’s function in $O(\log(N))$ modular multiplications by computing $X^{a'} \bmod N \equiv X^{a' \bmod \phi(N)} \bmod N$.

Observe that this puzzle is likewise multiplicative homomorphic: given a and a' , the product of the solutions $X^{a'}$ and $X^{a''}$ represents a solution for $a' + a''$. This preserves the homomorphic properties of the underlying POR and allows for batch verification for all the replicas and hence enables compact proofs.

To further reduce the verification burden on users, **Mirror** generates the exponents using a Linear Feedback Shift Registers (LFSR) as follows.

Linear Feedback Shift Registers: A *Linear Feedback Shift Register (LFSR)* is a popular building block for designing stream ciphers as it enables the generation of long output streams based on a initial state. In **Mirror**, LFSRs will be used to generate the exponents for the RSA-based puzzle described above. In what follows, we briefly describe the concept of an LFSR sequence and refer the readers to [29] for further details.

Definition 3 (Linear Feedback Shift Register)
Let \mathbb{F} be some finite field, e.g., \mathbb{Z}_p for some prime p . A Linear Feedback Shift Register (LFSR) of length λ consists of an internal state of length λ and a linear feedback function $F : \mathbb{F}^\lambda \rightarrow \mathbb{F}$ with $F(x_1, \dots, x_\lambda) = \sum_{i=1}^{\lambda} c_i \cdot x_i$. Given an initial state $(s_1, \dots, s_\lambda) \in \mathbb{F}^\lambda$, it defines inductively an LFSR sequence $(s_t)_{t \geq 1}$ by $s_{t+\lambda} = F(s_t, \dots, s_{t+\lambda-1})$ for $t \geq 1$.

An important and related notion is that of a *feedback polynomial*. Given an LFSR with feedback function $F(x_1, \dots, x_\lambda) = \sum_{i=1}^{\lambda} c_i \cdot x_i$, the feedback polynomial $f(x) \in \mathbb{F}[x]$ is defined as:

$$f(x) = x^\lambda - \sum_{i=1}^{\lambda} c_i \cdot x^{i-1}. \quad (2)$$

It holds that any multiple of a feedback polynomial is again a feedback polynomial. That is, if $f^*(x) = x^{\lambda^*} - \sum_{i=1}^{\lambda^*} c_i^* \cdot x^{i-1}$ is a multiple of f , then it holds that $s_{t+\lambda^*} - \sum_{i=1}^{\lambda^*} c_i^* \cdot s_{t+i-1} = 0$ for each $t \geq 1$. **Mirror** exploits this feature in order to realize a gap between the puzzle solution created by provider and the verification done by the user.

3.3 Protocol Specification

We now start by detailing the procedures in **Mirror**.

Specification of the Store Procedure: In the store phase, the user is interested in uploading a file $D \in \{0, 1\}^*$. We assume that the file D is encrypted to protect its confidentiality and encoded with an erasure code (as required by the utilized POR in order to provide extractability guarantees) prior to being input to the **Store** protocol [25, 34]. First, the user generates an RSA modulus $N := p \cdot q$ where p and q are two safe primes² whose size is chosen according to the security parameter κ .

Similar to [34], the file is interpreted as n blocks, each is s sectors long. A sector is an element of \mathbb{Z}_N and is denoted by $d_{i,j}$ with $1 \leq i \leq n$, $1 \leq j \leq s$. That is, the overall number of sectors in the file is $n \cdot s$. To ensure unique extractability (see Section 4.1), we require that the bit representation of each sector $d_{i,j}$ contains a characteristic pattern, e.g., a sequence of zero bits. The length of this pattern depends on the file size and should be larger than $\log_2(n \cdot s)$.

Furthermore, the user samples a key k_{prf} per file, where the key length is determined by the security parameter, e.g., $k_{\text{prf}} \in \{0, 1\}^\kappa$. By invoking k_{prf} as a seed to a pseudo-random function (PRF), the user samples s non-zero elements of $\mathbb{Z}_{\phi(N)}$, i.e., $\varepsilon_1, \dots, \varepsilon_s \xleftarrow{\text{R}} \mathbb{Z}_{\phi(N)} \setminus \{0\}$. Finally, the user computes σ_i for each i , $1 \leq i \leq n$, as follows:

$$\sigma_i \leftarrow \prod_{j=1}^s d_{i,j} \varepsilon_j \in \mathbb{Z}_N. \quad (3)$$

These values are appended to the original file so that the user uploads $(D, \{\sigma_i\}_{1 \leq i \leq n})$. Unless specified otherwise, we note that all operations are performed in the multiplicative group \mathbb{Z}_N^* of invertible integers modulo N .³

Assuming that the user is interested in maintaining r replicas in addition to the original file D at the cloud, the user additionally constructs copy parameters Π which will also be sent to the server. To this end, the user first generates two elements $g, h \in \mathbb{Z}_N^*$ of order p' and q' , respectively. Recall that the order of \mathbb{Z}_N^* is $\phi(N) = (p-1)(q-1) = 4 \cdot p' \cdot q'$. The elements g and h will be made public to the server while their orders are kept secret.

Then, the user proceeds to specify feedback polynomials for two LFSRs, one being defined over $\mathbb{Z}_{p'}$ and the other over $\mathbb{Z}_{q'}$. Both LFSRs need to have a length λ such that $|\mathbb{F}|^\lambda > n \cdot s$. Here, for each of the two LFSRs, *two* feedback polynomials are specified: a shorter one which will be kept secret by the user

²That is, $p-1 = 2 \cdot p'$ and likewise $q-1 = 2 \cdot q'$ for two distinct primes p' and q' .

³Observe that hitting by coincidence a value outside of \mathbb{Z}_N^* allows to factor N which is considered to be a hard problem.

and a larger one that will be made public to the provider. More precisely, for the LFSR defined over $\mathbb{Z}_{p'}$ the user chooses two polynomials

$$f_a(x) := x^\lambda - \sum_{i=1}^{\lambda} \alpha_i \cdot x^{i-1}, \quad f_a^*(x) := x^{\lambda^*} - \sum_{i=1}^{\lambda^*} \alpha_i^* \cdot x^{i-1}$$

such that $f_a^*(x)$ is a multiple of $f_a(x)$ (and hence $\lambda < \lambda^*$). For security reasons, it is necessary to ensure that $\alpha_1^* \geq 2$.

The feedback polynomial $f_a(x)$ with the lower degree will be kept secret while the polynomial $f_a^*(x)$ of the higher degree and the larger coefficients will be given to the provider. $f_a(x)$ will serve as a feedback polynomial to generate for each replica $k \in \{1, \dots, r\}$ an LFSR sequence $(a_t^{(k)})$. To this end, the user chooses for each k an initial state $(a_1^{(k)}, \dots, a_\lambda^{(k)}) \in \mathbb{Z}_{p'}^\lambda$ which defines the full sequence by $a_{t+\lambda+1}^{(k)} = \sum_{i=1}^\lambda \alpha_i \cdot a_{t+i}^{(k)}$ for any $t \geq 0$. Observe that due to the fact that $f_a^*(x)$ is a multiple of $f_a(x)$, it likewise holds $a_{t+\lambda^*+1}^{(k)} = \sum_{i=1}^{\lambda^*} \alpha_i^* \cdot a_{t+i}^{(k)}$ for any $t \geq 0$. Finally, the user publishes as part of the copy parameters the values $g^{a_1^{(k)}}, \dots, g^{a_{\lambda^*}^{(k)}} \in \mathbb{Z}_N$ for each replica and the coefficients $\alpha_1^*, \dots, \alpha_{\lambda^*}^* \in \mathbb{Z}$. Afterwards, he proceeds analogously over $\mathbb{Z}_{q'}$, i.e., sample coefficients $\beta_i \in \mathbb{Z}_{q'}$, compute feedback functions $f_b(x), f_b^*(x)$, choose an initial state $(b_1^{(k)}, \dots, b_\lambda^{(k)})$ for each replica, and so on.

Summing up, and assuming that the server should construct r replicas, the user sets the file specific verification tag (which are kept secret by the user) to:

$$\tau := \left(k_{\text{prf}}, p, q, g, h, (a_1^{(k)}, \dots, a_\lambda^{(k)})_{1 \leq k \leq r}, (\alpha_1, \dots, \alpha_\lambda), (b_1^{(k)}, \dots, b_\lambda^{(k)})_{1 \leq k \leq r}, (\beta_1, \dots, \beta_\lambda) \right).$$

To enable the server to construct the r replicas, the following copy parameters are given to the server:

$$\Pi := \left((g^{a_1^{(k)}}, \dots, g^{a_{\lambda^*}^{(k)}})_{1 \leq k \leq r}, (\alpha_1^*, \dots, \alpha_{\lambda^*}^*), (h^{b_1^{(k)}}, \dots, h^{b_{\lambda^*}^{(k)}})_{1 \leq k \leq r}, (\beta_1^*, \dots, \beta_{\lambda^*}^*) \right).$$

That is, the user sends D , the values $\{\sigma_i\}_{1 \leq i \leq n}$, and Π to the service provider and keeps the verification tag τ secret. Observe that the size of τ and Π are independent of the file size.

Specification of the CheckReplica Procedure: As the replicas are completely generated by the service provider, a **CheckReplica** procedure is not required in Mirror. However, one could check the validity of the

data replicas by running the **Replicate** procedure and simply comparing the outputs.

Specification of the Replicate Procedure: Upon reception of D , the values $\{\sigma_i\}_{1 \leq i \leq n}$, and Π , the service provider \mathcal{S} stores D and starts the construction of the r additional replicas $D^{(k)}$ for $1 \leq k \leq r$, of D . Here, each sector $d_{i,j}^{(k)}$ of replica k has the following form:

$$d_{i,j}^{(k)} = d_{i,j} \cdot g_{i,j}^{(k)} \cdot h_{i,j}^{(k)}, \quad (4)$$

We call these values $g_{i,j}^{(k)}$ and $h_{i,j}^{(k)}$ *blinding factors*. Both sets of blinding factors are computed by raising g and h by elements of the LFSR sequences $a_t^{(k)}$ and $b_t^{(k)}$, respectively, but one in the forward and the other in the backward order, namely:

$$g_{i,j}^{(k)} := g^{a_{(i-1)\cdot s+j}^{(k)}}, \quad h_{i,j}^{(k)} := h^{b_{(n\cdot s+1)-(i-1)\cdot s-j}^{(k)}}. \quad (5)$$

To enable the provider to compute the blinding factors $g^{a_i^{(k)}}$, we make use of the fact that for any $t \geq 0$ it holds $a_{t+\lambda^*+1}^{(k)} = \sum_{i=1}^{\lambda^*} \alpha_i^* \cdot a_{t+i}^{(k)}$ and hence

$$g^{a_{t+\lambda^*+1}^{(k)}} = \prod_{i=1}^{\lambda^*} \left(g^{a_{t+i}^{(k)}} \right)^{\alpha_i^*}. \quad (6)$$

The computation of the blinding factors $h^{b_i^{(k)}}$ works analogously.

In summary, the server constructs replicas $D^{(k)}$ for $k = 1, \dots, r$ as follows:

$$\begin{pmatrix} d_{1,1} \cdot g^{a_1^{(k)}} \cdot h^{b_{(n-1)\cdot s+s}^{(k)}} & \dots & d_{1,s} \cdot g^{a_s^{(k)}} \cdot h^{b_{(n-1)\cdot s+1}^{(k)}} \\ d_{2,1} \cdot g^{a_{s+1}^{(k)}} \cdot h^{b_{(n-2)\cdot s+s}^{(k)}} & \dots & d_{2,s} \cdot g^{a_{2,s}^{(k)}} \cdot h^{b_{(n-2)\cdot s+1}^{(k)}} \\ \vdots & \ddots & \vdots \\ d_{n,1} \cdot g^{a_{(s-1)\cdot s+1}^{(k)}} \cdot h^{b_s^{(k)}} & \dots & d_{n,s} \cdot g^{a_{n,s}^{(k)}} \cdot h^{b_1^{(k)}} \end{pmatrix}$$

Specification of the Verify Procedure: The **Verify** protocol generates at first a random challenge C . It contains a random ℓ -element set of tuples (i, v_i) where $i \in \{1, \dots, n\}$ denotes a block index, and $v_i \stackrel{\text{R}}{\leftarrow} \mathbb{Z}_N$ is a randomly generated integer. In addition, a non-zero subset $R \subset \{1, \dots, r\}$ is sampled. The set R will indicate which replicas will be involved in the challenge. Observe that $R = \emptyset$ would mean that simply a proof of retrievability is executed without checking the replicas. The challenge is then the combination of both:

$$C = ((i_c, v_c)_{c=1}^\ell, R). \quad (7)$$

Given a challenge C , the server computes the response $\mu = (\mu_1, \dots, \mu_s) \in \mathbb{Z}_N^s$ as follows:

$$\mu_j := \prod_{c=1}^\ell d_{i_c,j}^{v_c}, \quad j = 1, \dots, s. \quad (8)$$

Observe that μ_j is the product of powers of the original data, that is $d_{i_c,j}^{v_c}$. In addition, the file tags are processed in the same manner to obtain:

$$\sigma = \prod_{c=1}^{\ell} \left(\sigma_{i_c} \cdot \prod_{j=1}^s \prod_{k \in R} d_{i_c,j}^{(k)} \right)^{v_c}. \quad (9)$$

The tuple (μ, σ) marks the response and is sent back to the user who verifies (μ, σ) similar to the private-verifiable POR of [34]. First, he computes:

$$\tilde{\sigma} := \sigma \cdot \prod_{c=1}^{\ell} \left(\prod_{j=1}^s \prod_{k \in R} g_{i_c,j}^{(k)} h_{i_c,j}^{(k)} \right)^{-v_c} \quad (10)$$

Observe that this will require the reconstruction of the blinding factors. In Appendix C, we show how to efficiently perform this verification by the user, using the knowledge of the verification tags—in particular the knowledge of the secret shorter LFSR, its initial state, and the order of g and h , respectively.

Next, the user recovers the secret parameters ε_{i_k} , $k = 1, \dots, \ell$, using the key k_{prf} as a seed for the PRF. Finally, the user verifies that the following holds:

$$\prod_{j=1}^s \mu_j^{\varepsilon_j + |R|} = \tilde{\sigma}. \quad (11)$$

We now explain why the verification step has to hold if the response has been computed correctly. First, it follows from Equation (4) that $\prod_{c=1}^{\ell} \left(\prod_{j=1}^s \prod_{r \in R} d_{i_c,j}^{(r)} \right)^{v_c}$ can be rewritten as the product of $\prod_{c=1}^{\ell} \left(\prod_{j=1}^s \prod_{r \in R} d_{i_c,j} \right)^{v_c}$ and $\prod_{c=1}^{\ell} \left(\prod_{j=1}^s \prod_{r \in R} g_{i_c,j}^{(r)} h_{i_c,j}^{(r)} \right)^{v_c}$. The second factor is exactly the part that is canceled out in Equation (10) while the first factor can be simplified to $\prod_{c=1}^{\ell} \left(\prod_{j=1}^s d_{i_c,j}^{|R|} \right)^{v_c}$.

Given a series of straightforward calculations, one can show that $\tilde{\sigma}$ can be rewritten to $\prod_{j=1}^s (\mu_j)^{\varepsilon_j + |R|}$. This proves the correctness of Equation (11)—hence the correctness with respect to extractability.

Moreover, it is easy to see that, since each sector of a replica corresponds to the multiplication of the corresponding sector of the uploaded file D with a blinding factor (that can be reconstructed from Π), the replicas are indeed copies of the original file. This means that **Replicate** is correct. Summing up, all three correctness requirements explained in Section 2 are fulfilled in **Mirror**.

4 Security Analysis

We now proceed to prove the security of our scheme according to the definitions in Section 2.4. Recall that

the user is not involved in the replica generation and that the size of the parameters involved in creating a replica is independent of the file size. This ensures the correct replication property described in Section 2.4.

It remains to prove that (i) if the service provider \mathcal{S} stores at least a fraction δ of all sectors in one replica, then the file can be reconstructed (extractability) and (ii) if the service provider stores less than a fraction of δ of any replica, this misbehavior will be detected with overwhelming probability (storage allocation).

4.1 Extractability

In principle, the computations done in the **Store** and **Verify** procedures of **Mirror** can be seen as multiplicative variants of the corresponding mechanisms of the privately-verifiable POR of [34] (see Appendix B for details on the scheme of [34]). In particular, the extractability arguments given in [34] transport directly to **Mirror**. We assume that an erasure coding is applied to the file to ensure the recovery of file contents from any fraction δ of the file. In particular, we refer to [34] for additional details on the choice of parameters (e.g., for erasure coding) such that retrievability is ensured if a fraction δ of the file is stored.

To show that **Mirror** enables the reconstruction of the file from sufficiently many correct responses, we point out that given a correct response, the user learns expressions of the form $\mu_j = \prod_{c=1}^{\ell} d_{i_c,j}^{v_c}$ for known exponents $v_c \in \mathbb{Z}$ and known indices. Let us assume some arbitrary ordering $\mu^{(1)}, \mu^{(2)}, \dots$ on these expressions. If sufficiently many responses $\mu^{(i)}$ are known, the user can choose for any (i, j) coefficients $c^{(k)} \in \mathbb{Z}$ such that $\prod_k (\mu^{(k)})^{c^{(k)}} = d_{i,j}^u =: \tilde{d}$ for a known value $u \in \mathbb{Z}$.

Recall that the order of any $d_{i,j} \in \mathbb{Z}_N^*$ is a divisor of $2p'q'$. If u is odd, u is co-prime to $p'q'$ with overwhelming probability and the user can simply compute $u^{-1} \bmod p'q'$ and determine $d_{i,j} = \tilde{d}^{u^{-1}}$. On the other hand, if u is even (i.e., $u = 2 \cdot u'$), two cases emerge. If the order of $d_{i,j}$ is a divisor of $p'q'$, the exponent is again co-prime to the order with high probability. In this case, the user computes $u^{-1} \bmod p'q'$ and checks if $\tilde{d}^{u^{-1}}$ contains the characteristic bit pattern (see description of the **Store** procedure). If this fails, this means that the order of \tilde{d} is even and the user proceeds as follows. Observe that the order of $d_{i,j}^2$ is a divisor of $p'q'$. Thus, the user first computes $(u')^{-1} \bmod p'q'$ and then $\tilde{d}^{(u')^{-1}}$. This yields $d_{i,j}^2$. As the user knows the factorization of N , he can compute all four possible roots of $d_{i,j}^2$.

(e.g., using the Chinese Remainder Theorem). Due to the characteristic pattern embedded in $d_{i,j}$ (see the specification of the `Store` procedure), the user is able to identify the correct $d_{i,j}$.⁴

4.2 Storage Allocation

Observe that `Mirror` represents in fact a proof of retrievability over the uploaded file *and* all replicas. This means that if a challenge involves sectors that are not stored, `Mirror` ensures that the service provider fails with high probability unless he is able to correctly reconstruct the missing replicas. In the following, we therefore investigate the effort of a malicious service provider in reconstructing missing sectors. That is, we consider the scenario where the service provider has stored the complete file⁵ but only parts of some replicas.

In `Mirror`, the service provider needs to compute the corresponding blinding factors in order to recompute any missing sectors. As these are products of the form $g_i^{(k)} \cdot h_j^{(k)}$ and since these sequences are independent from each other, the service provider is forced to store values of the sequences $(g_i^{(k)})_i$ and $(h_j^{(k)})_j$ separately. Moreover, since these values are different for the individual replicas, knowing (or reconstructing) a value from one sequence and one replica does not help the service provider in deriving values of other sequences and/or replicas.

A crucial aspect of `Mirror` is that a cheating service provider should require a significantly higher effort compared to an honest service provider in recomputing missing replicas. Recall that both the user and the provider determine the blinding factors by computing LFSR sequences. One difference though is that the provider has to do his computations on values $g_i^{a_i^{(k)}}, h_j^{b_j^{(k)}} \in \mathbb{Z}_N^*$ while the user can efficiently compute on the exponents in $a_i^{(k)} \in \mathbb{Z}_{p'}$ and $b_j^{(k)} \in \mathbb{Z}_{q'}$ directly. Observe that the provider is not able to transfer his computations into $\mathbb{Z}_{p'}$ and $\mathbb{Z}_{q'}$ without eventually factoring $N = p \cdot q$, which is commonly assumed to be a hard problem. A further gap is that the user deploys LFSRs with feedback functions $f_a(x) \in \mathbb{Z}_{p'}[x]$ and $f_b(x) \in \mathbb{Z}_{q'}$ where the provider only knows the feedback functions $f_a^*(x), f_b^*(x) \in \mathbb{Z}[x]$ that are multiples of $f_a(x), f_b(x)$. Given that these functions involve more inputs, and require the construc-

⁴Observe that in principle it may happen with some probability that more than one root exhibits this pattern. This is the reason why a padding length $\geq \log_2(n \cdot s)$ is proposed such that the expected number of incorrectly reconstructed sectors in the file is less than 1.

⁵Recall that this property is validated by the proof of retrievability already.

tion of larger coefficients, this would incur additional (significant) computational overhead on the provider compared to the user. For deducing the shorter feedback functions $f_a(x), f_b(x)$, the provider would have to determine $\mathbb{Z}_{p'}$ respectively $\mathbb{Z}_{q'}$ —which equally require the knowledge of the factors of N .

It remains to investigate the effort for reconstructing values of the LFSR sequences. Note that the sequences used in the replicas are defined by different independent internal states and that, for each replica, the sequences $(a^{(k)i})$ and $(b_j^{(k)})$ are independent. We can therefore without loss of generality restrict our analysis to one sequence $(g_i)_{i \geq 1}$. For the ease of representation, we omit in the sequel the index (k) and write $g_i = g^{a_i}$ for the ease of representation. We say that $\vec{v} = (v_1, \dots, v_{n \cdot s}) \in \mathbb{Z}^{n \cdot s}$ represents a *valid relation* with respect to $(g_i)_{i \geq 1}$ if:

$$\prod_{i=1}^{n \cdot s} g_i^{v_i} = 1. \quad (12)$$

It follows from known facts about LFSRs that valid relations are the only means for the service provider to compute missing values g_j from known values g_i (see Appendix D for more details).

As the provider is forced to use the feedback function defined by the coefficients $(\alpha_1^*, \dots, \alpha_{\lambda^*}^*)$ (see above), the only valid relations the provider can derive are linear combinations of:

$$\vec{b}_j := (\underbrace{0 \dots 0}_{j-1}, \alpha_1^*, \dots, \alpha_{\lambda^*}^*, \underbrace{0 \dots 0}_{n \cdot s - (j-1) - \lambda^*}), \quad (13)$$

where \vec{b}_1 corresponds to the given feedback polynomial $f_a^*(x)$ and the others are derived by simple shift of indexes.

Hence, for any valid relation $\vec{v} = (v_i)_i \in \mathbb{V}$, there exist unique coefficients $c_1, \dots, c_{n \cdot s - \lambda^* + 1} \in \mathbb{Z}$ such that $\vec{v} = \sum_i c_i \cdot \vec{b}_i$. Let i_{\min} be the smallest index with $c_{i_{\min}} \neq 0$. Then, it holds that the first $v_j = 0$ for $j < i_{\min} - 1$ and that $v_{i_{\min}} = c_{i_{\min}} \cdot \alpha_1^*$ (as the other vectors with index $i > i_{\min}$) are zero at index i_{\min} . Hence, it holds that $\max_i \{\lceil \log_2(v_i) \rceil\} \geq \lceil \log_2(\alpha_1^*) \rceil$.

This shows that the effort of *executing* a valid relation (cf. Equation (12)) involves at least one exponentiation with an exponent of size $\lceil \log_2(\alpha_1^*) \rceil$.⁶ The effort to compute one exponentiation for an exponent of bitsize k is $3/2 \cdot k \cdot T_{\text{mult}}$, where T_{mult} stands for the time it takes the resource-constrained rational attacker (see Section 2.3) to multiply two values modulo N [27].

⁶One may combine several exponentiations to reduce the overall number of exponentiations but one cannot reduce the effort to compute at least once the exponentiation with the highest exponent.

Thus, a pessimistic lower bound for reconstructing a missing value g_i is $3/2 \cdot \lceil \log_2(\alpha_1^*) \rceil \cdot T_{\text{mult}}$. Observe that we ignore here additional efforts such as finding appropriate valid relations (cf. Equation (12)), etc.

Assume now that the service provider stores less than a fraction δ of all sectors of a given replica where δ refers to the threshold chosen by the user (see also Definition 2). Thus, for any value g_i contained in the challenge, the probability that g_i has to be re-computed is at least $1 - \delta$. Due to the fact that this holds for the values h_j as well and that a challenge requests $\ell \cdot s$ sectors, the expected number of values that need to be recomputed is $2\ell \cdot s \cdot (1 - \delta)$. To achieve the binding property with respect to a rational attacker, one has to ensure that the time effort for recomputing these values incurs costs that exceed the costs for storing these values. This implies a time threshold T_{thr} which marks the minimum computational effort this should take. Given such a threshold T_{thr} , we get the following inequality:

$$\lceil \log_2(\alpha_1^*) \rceil \geq \frac{T_{\text{thr}}}{3\ell \cdot s \cdot (1 - \delta) \cdot T_{\text{mult}}}. \quad (14)$$

That is, if the parameters are chosen as displayed in (14), a dishonest provider would bear on average higher costs than an honest provider. Here, one can use the common cut-and-choose approach, by posing a number of challenges where the number is linear in the security parameter κ , to ensure that the overall probability to circumvent these computations is negligible in κ . This proves the binding property (cf. Definition 2) with respect to the class of PPT service providers that can execute a bounded number of threads in parallel only.

Notice that Mirror can easily cope with (i) different attacker strengths, and (ii) variable cost metrics, as follows:

Length of LFSR: One option would be to increase λ^* , i.e., the length of the LFSR communicated to the provider, and hence the number of values $g^{a_i^{(k)}}$ and $h^{b_j^{(k)}}$ the provider has to use for generating the replicas. In the extreme case, λ^* could be made equal to half of the total number of sectors $n \cdot s$ —which would result into a scheme whose bandwidth consumption is comparable to [18].

Bitlength of coefficients: Another alternative would be to keep λ^* short, but to increase the bitlengths of the coefficients α_i^* and β_j^* . This would preserve the small bandwidth consumption of Mirror but would increase the time effort to run Replicate. This option will

also not affect the latency borne by users in verifying the provider’s response.

Hybrid approach: Clearly, one can also aim for a hybrid scheme, by increasing both the public LFSR length λ^* and the coefficients α_i^* and β_j^* .

In Section 5, we investigate reasonable choices of α_1^* , ρ , and T_{thr} to satisfy Equation 14. Of course, the same considerations can also be applied with respect to β_1^* which we omit for space reasons.

5 Implementation & Evaluation

In this section, we evaluate an implementation of Mirror within a realistic cloud setting and we compare the performance of Mirror to the MR-PDP solution of [18].

5.1 Implementation Setup

We implemented a prototype of Mirror in Scala. For a baseline comparison, we also implemented the multi-replica PDP protocol⁷ of [18], which we denote by MR-PDP in the sequel (see Appendix A for a description of the MR-PDP of [18]). In our implementation, we relied on SHA-256, and the Scala built-in random number generator.

We deployed our implementation on a private network consisting of two 24-core Intel Xeon E5-2640 with 32GB of RAM. The storage server was running on one 24-core Xeon E5-2640 machine, whereas the clients and auditors were co-located on the second 24-core Xeon E5-2640 machine.

To emulate a realistic Wide Area Network (WAN), the communication between various machines was bridged using a 100 Mbps switch. All traffic exchanged on the networking interfaces of our machines was shaped with NetEm [31] to fit a Pareto distribution with a mean of 20 ms and a variance of 4 ms—thus emulating the packet delay variance specific to WANs [19].

In our setup, each client invokes an operation in a closed loop, i.e., a client may have at most one pending operation.

When implementing Mirror, we spawned multiple threads on the client machine, each thread corresponding to a unique worker handling a request of a client. Each data point in our plots is averaged over 10 independent measurements; where appropriate, we include the corresponding 95% confidence intervals.

⁷We acknowledge that PDP offers weaker guarantees than POR. However, to the best of our knowledge, no prior proposals for multi-replica-POR exist. MR-PDP thus offers one-of-the-few reasonable benchmarks for Mirror.

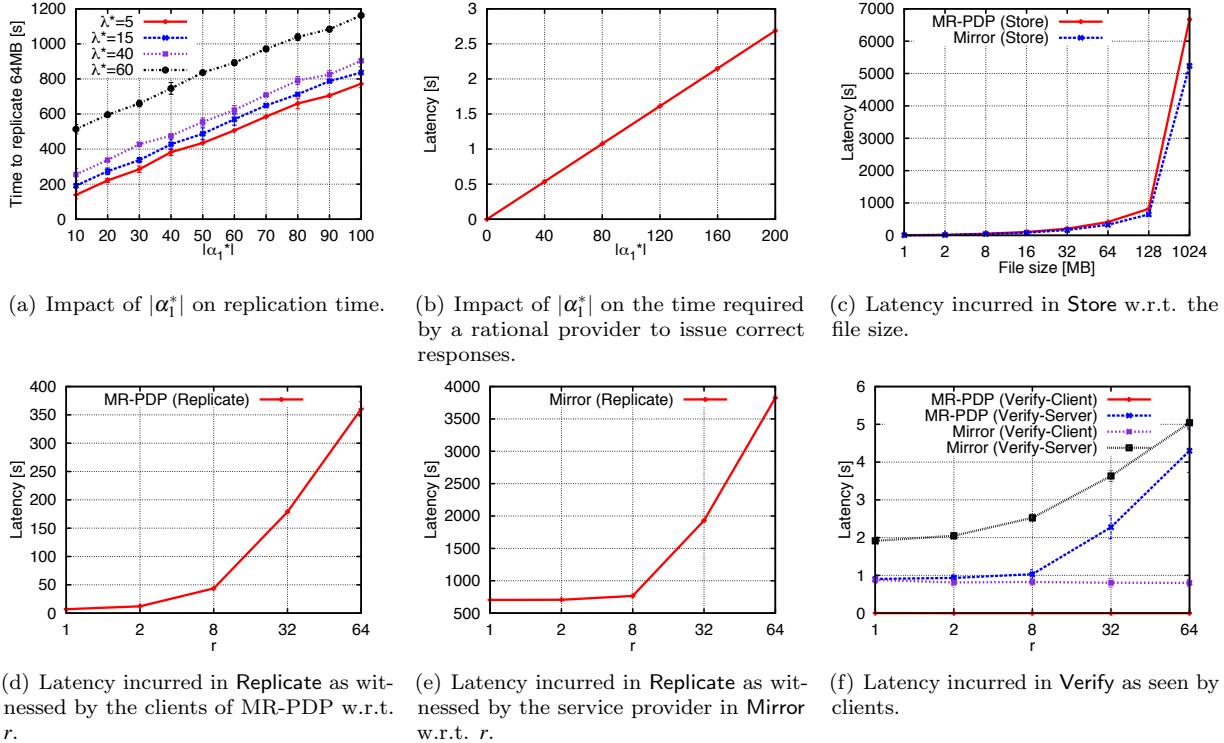


Figure 1: Performance evaluation of Mirror in comparison to the MR-PDP scheme of [18]. Each data point in our plots is averaged over 10 independent runs; where appropriate, we also show the corresponding 95% confidence intervals.

Parameter	Default Value
File size	64 MB
$ p $	1024 bits
$ q $	1024 bits
RSA modulus size	2048 bit
Number of challenges ℓ	40 challenges
Length of secret LFSR λ	2
Length of public LFSR λ^*	15
Fraction of stored sectors δ	0.9
Number of replicas r	2

Table 2: Default parameters used in the evaluation.

Table 2 summarizes the default parameters assumed in our setup.

5.2 Evaluation Results

Before evaluating the performance of Mirror, we start by analyzing the impact of the block size on the latency incurred in the verification of Mirror and in MR-PDP. Our results (Figure 4 in Appendix E) show that modest block sizes of 8 KB yield the most balanced performance, on average, across the investigated schemes. In the rest of our evaluation, we therefore set the block size to 8 KB.

Impact of the bitsize of α_1^* : In our implementation, our choice of parameters was mainly governed by the need to establish a tradeoff between the replication performance and the resource penalty incurred on a dishonest provider. To this end, we choose a small value for the public LFSR length λ^* , i.e., the LFSR length communicated to \mathcal{S} , and small coefficients α_i^* and β_j^* (these coefficients were set to 1 bit for $i, j > 1$). Recall that using smaller coefficients allows for faster exponentiations and hence a decreased replication effort.

However, as shown in Equation 14, the bitsize of α_1^* (which we shortly denote by $|\alpha_1^*|$ in the following) plays a paramount role in the security of Mirror. Note that the same analysis applies to β_1^* —which we do not further consider to keep the discussion short. Clearly, $|\alpha_1^*|$ (and λ^*) also impacts the file replication time at the service provider. In Figure 1(a), we evaluate the impact of $|\alpha_1^*|$ on the replication time, and on the time invested by a rational provider (who does not replicate) to answer every client challenge in Mirror. Our results indicate that the file replication time grows linearly with $|\alpha_1^*|$. Moreover, the higher λ^* is, the longer it takes to replicate a given file. On the other hand, as shown in Figure 1(b), $|\alpha_1^*|$ consid-

$ \alpha_1^* $	Estimated EC2 costs per challenge (USD)
40	0.000058
70	0.00011
80	0.00013
120	0.00019

Table 3: Costs borne by a rational provider who computes the responses to a challenge of size $l = 40$ on the fly. We assume two replicas of size 64 MB, and estimate costs for a compute-optimized (extra large) instance from Amazon EC2 (at 0.42 USD per hour).

erably affects the time incurred on a rational provider which did not correctly replicate (some) user files. The larger $|\alpha_1^*|$ is, the longer it takes a misbehaving provider to reply to the user challenges, and thus the bigger are the amount of computational resources that the provider needs to invest in. Here, we assume the lower bound on the effort of a misbehaving provider (i.e., which only stores a fraction δ of the sectors per replica) given by Equation 14.

Setting $|\alpha_1^*|$: Following this analysis, suitable choices for α_1^* need to be large enough such that the costs borne by a rational provider who computes the responses on the fly are higher than those borne by an honest provider who correctly stores the replicas. In Table 3, we display the corresponding costs borne by a rational provider who computes the responses on the fly to a single challenge, assuming $l = 40$, and $r = 2$ replicas of size 64 MB. Here, we estimate the computation costs as follows: we interpolate the time required by a rational provider in answering challenges from Figure 1(c). We then estimate the corresponding computation costs assuming a compute-optimized (extra large) instance from Amazon EC2 (at 0.42 USD per hour), which offers comparable computing power than that used in our implementation setup.

For comparison purposes, notice that the cost of storing two 64 MB replicas per day (based on Amazon S3 pricing [9]) is approximately 0.00011 USD. *This shows that when instantiating Mirror with parameters $|\alpha_1^*| = 70$, the provider should not gain any (rational) advantage in misbehaving, if the user issues at least one PoR² challenge of $l = 40$ randomly selected blocks per day.* Clearly, users can increase the number of challenges that they issue accordingly to ensure that the costs borne by a rational provider are even more pronounced, e.g., to account for possible fluctuations in costs.

Following this analysis, we assume that $|\alpha_1^*| = 70$ throughout the rest of the evaluation. As shown in

Figure 1(a), this parameter choice results in reasonable replication times. e.g., when $\lambda^* = 5$ or $\lambda^* = 15$. Observe that, in this case, users can *detect/suspect* misbehavior by observing the cloud’s response time. As shown in Figure 1(f), the typical response time of an honest service provider is less than 2 seconds when $r = 2$. An additional 0.9 seconds of delay (i.e., totalling 2.9 seconds) in responding to a challenge can be then detected by users.

Store performance: In Figure 1(c), we evaluate the latency incurred in Store with respect to the file size. Our findings suggest that the Store procedure of Mirror is considerably faster than that of MR-PDP. This is the case since the tag creation in Mirror requires fewer exponentiations per block (cf. Appendix A). For instance, the Store procedures is almost 20% faster than that of MR-PDP for files of 64MB in size.

Replicate performance: Figure 1(d) depicts the latency incurred on the clients of MR-PDP in the replicate procedure Replicate with respect to the number of replicas. Recall that, in MR-PDP, clients have to process and upload all replicas by themselves to the cloud. Clearly, the latency of Replicate increases with the number of replicas stored. Given our multi-threaded implementation, notice that the replication process can be performed in parallel. Here, as the number of concurrent replication requests increases, the threads in our thread pool are exhausted and the system saturates—which explains the sharp increase in the latency witnessed by clients who issue more than 8 concurrent replication requests. Notice that users of Mirror do not bear any overhead due to replication since this process is performed by the service provider.

In Figure 1(e), we show the latency incurred on the service provider in Mirror with respect to the number of replicas r . Since Mirror relies on puzzles, the replication process consumes considerable resources from the service provider. However, we point out that is a one-time effort per file, and can be performed offline (i.e., the provider can replicate files using “offline” resources in the back-end). For example, the creation of 8 additional 64 MB file replicas incurs a latency of almost 765 seconds. As mentioned earlier, Mirror trades this additional computational burden with bandwidth. Namely, users of Mirror only have to upload the file once, irrespective of the number of replicas desired. This, in turn, reduces the download bandwidth of providers and, as a consequence, the costs of offering the service.

In Figure 2, we estimate the costs of the additional computations incurred in Mirror for a 64 MB file,

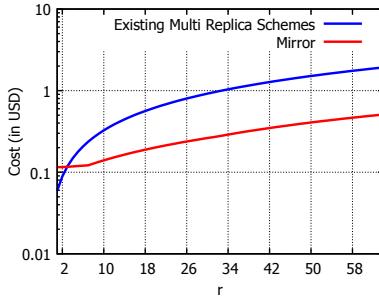


Figure 2: Replication costs for a 64 MB file (in USD) incurred **Mirror** vs. existing multi-replica schemes. We assume that the provider provisions a large general instance from Amazon EC2 at 0.441 USD per hour). We assume the replication time given by our experiments in Figure 1(e) and we estimate bandwidth costs by adapting the findings of [3] (cf. Table 3).

compared to those incurred by existing multi-replica schemes which require users to upload all the replicas. To estimate computing costs, we rely on the AWS pricing model [8]; we assume that the provider provisions a multi-core (compute-optimized) extra large instance from Amazon EC2 (at 0.441 USD per hour). We rely on our findings in Figure 1(e) to estimate the computing time for replication. We estimate bandwidth costs by adapting the findings of [3] (i.e., by assuming \$5 per Mbps per month cf. Table 3). Our estimates suggest that **Mirror** considerably reduces the costs borne on the provider by trading bandwidth costs with the relatively cheaper computing costs. We expect that the cost savings of **Mirror** will be more significant for larger files, and/or additional replicas.

Verify performance: In Figure 1(f), we evaluate the latency witnessed by the users and service provider in the Verify procedure of **Mirror** and MR-PDP, respectively. Our findings show that the verification overhead witnessed by the service provider in **Mirror** is almost twice that of MR-PDP. Moreover, users of **Mirror** require almost 1 second to verify the response issued by the provider. Notice that the majority of this overhead is spent while computing/verifying the response to the issued challenge. This discrepancy mainly originates from the fact that the challenge-response in **Mirror** involves all the 32 sectors of each block in order to ensure the extractability of all replicas⁸. We contrast this to MR-PDP where each block comprises a single sector—which only ensures data/replica possession but does not provide

⁸We point out that this is not particular to **Mirror** and applies to all schemes which ensure retrievability (e.g., [34]).

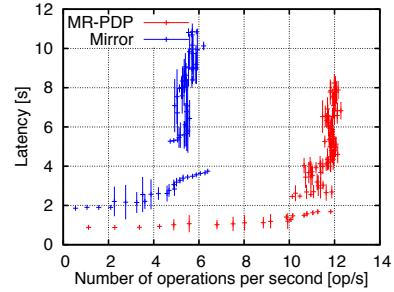


Figure 3: Latency vs. throughput comparison between MR-PDP and **Mirror**.

extractability guarantees.

In Figure 3, we evaluate the peak throughput exhibited by the service provider in the Verify procedure. Here, we require that the service provider handles verification requests back to back; we then gradually increase the number of requests in the system (until the throughput is saturated) and measure the associated latency. Our results confirm our previous analysis and show that **Mirror** attains a maximum throughput of 6 verification operations per second; on the other hand, the service provider in MR-PDP can handle almost 12 operations per second. As mentioned earlier, this discrepancy is mainly due to the fact that the MR-PDP blocks only comprise a single sector, whereas each block in **Mirror** comprises 32 sectors. However, we argue that the overhead introduced by our scheme compared to MR-PDP can be easily tolerated by clients; for instance, for 64 MB files, our proposal only incurs an additional latency overhead of 800 ms on the clients when compared to MR-PDP.

6 Related Work

Curtmola *et al.* propose in [18] a multi-replica PDP, which extends the basic PDP scheme in [12] and enables a user to verify that a file is replicated at least across t replicas by the cloud. In [16], Bowers *et al.* propose a scheme that enables a user to verify if his data is stored (redundantly) at multiple servers by measuring the time taken for a server to respond to a read request for a set of data blocks. In [13, 14], Barsoum and Hasan propose a multi-replica dynamic data possession scheme which allows users to verify multiple replicas, and to selectively update/insert their data blocks. This scheme builds upon the BLS-based SW scheme of [34]. In [22], the authors extend the dynamic PDP scheme of [21] to transparently support replication in distributed cloud storage systems. All existing schemes however share a common system model, where the user constructs and uploads the

replicas onto the cloud. On the other hand, **Mirror** conforms with the existing cloud model by allowing users need to process/upload their original files only once irrespective of the replication performed by the cloud provider.

Proofs of location (PoL) [32, 36] aim at proving the geographic position of data, e.g., if it is stored on servers within a certain country. In [36], Watson *et al.* provide a formal definition for PoL schemes by combining the use of geolocation techniques together with the SW POR schemes [34]. In [36], the authors assume a similar system model to **Mirror**, where the user uploads his files to the service provider only once. The latter then re-codes the tags of the file, and replicates content across different geo-located servers. Users can then execute individual PORs with each server to ensure that their data is stored in its entirety at the desired geographical location. We contrast this to our solution, where the user has to invoke a single **Mirror** instance to efficiently verify the integrity of all stored replicas.

Proofs of space [20] ensure that a prover can only respond correctly if he invests at least a certain amount of space or time per execution. However, this notion is not applicable to our scenario where we need to ensure that a *minimum* amount of space has been invested by the prover. Moreover, the instantiation in [20] does not support batch-verification which is essential in **Mirror** to conduct POR on several replicas in a single protocol run.

7 Conclusion

In this paper, we proposed a novel solution, **Mirror**, which enables users to efficiently verify the retrievability of their data replicas in the cloud. Unlike existing schemes, the cloud provider replicates the data by itself in **Mirror**; by doing so, **Mirror** trades expensive bandwidth resources with cheaper computing resources—a move which is likely to be welcomed by providers and customers since it promises better service while lowering costs.

Consequently, we see **Mirror** as one of the few economically-viable and workable solutions that enable the realization of verifiable replicated cloud storage.

8 Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback and comments. This work was partly supported by the TREDISEC project (G.A. no 644412), funded by the European

Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme.

References

- [1] Amazon S3 Introduces Cross-Region Replication.
- [2] Cloud Computing: Cloud Security Concerns. <http://technet.microsoft.com/en-us/magazine/hh536219.aspx>.
- [3] The Relative Cost of Bandwidth Around the World.
- [4] Amazon S3 Service Level Agreement, 2009. <http://aws.amazon.com/s3-sla/>.
- [5] Are We Safeguarding Social Data?, 2009. MIT Technology Review, <http://www.technologyreview.com/view/412041/are-we-safeguarding-social-data/>.
- [6] Microsoft Corporation. Windows Azure Pricing and Service Agreement, 2009.
- [7] Protect data stored and shared in public cloud storage. http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell_Data_Protection_Cloud_Edition_Data_Sheet.pdf, 2013.
- [8] Amazon EC2 Pricing, 2015. <https://aws.amazon.com/ec2/pricing/>.
- [9] Amazon S3 Pricing, 2015. http://aws.amazon.com/s3/pricing/?nc2=h_ls.
- [10] Google loses data after lightning strikes. <http://money.cnn.com/2015/08/19/technology/google-data-loss-lightning/>, 2015.
- [11] Frederik Armknecht, Jens-Matthias Bohli, Ghassem O. Karame, Zongren Liu, and Christian A. Reuter. Outsourced proofs of retrievability. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’14, pages 831–843, New York, NY, USA, 2014. ACM.
- [12] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Xiaodong Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security*, pages 598–609, 2007.
- [13] Ayad F. Barsoum and M. Anwar Hasan. Integrity verification of multiple data copies over untrusted cloud servers. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012*, pages 829–834, 2012.
- [14] Ayad F. Barsoum and M. Anwar Hasan. Provable multicopy dynamic data possession in cloud computing systems. *IEEE Transactions on Information Forensics and Security*, 10(3):485–497, 2015.

- [15] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security*, pages 187–198, 2009.
- [16] Kevin D. Bowers, Marten van Dijk, Ari Juels, Alina Oprea, and Ronald L. Rivest. How to tell if your cloud files are vulnerable to drive crashes. In *ACM Conference on Computer and Communications Security*, pages 501–514, 2011.
- [17] Jin-yi Cai, Richard J. Lipton, Robert Sedgewick, and Andrew Chi-Chih Yao. Towards uncheatable benchmarks. In *Proceedings of the Eighth Annual Structure in Complexity Theory Conference, San Diego, CA, USA, May 18-21, 1993*, pages 2–11, 1993.
- [18] Reza Curtmola, Osama Khan, Randal C. Burns, and Giuseppe Ateniese. MR-PDP: Multiple-Replica Provable Data Possession. In *ICDCS*, pages 411–420, 2008.
- [19] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. Powerstore: Proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS ’13*, pages 285–298, New York, NY, USA, 2013. ACM.
- [20] Stefan Dziembowski, Sebastian Faust, Vladimir Kostrov, and Krzysztof Pietrzak. Proofs of space. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 585–605. Springer, 2015.
- [21] C. Christopher Erway, Alptekin Küpcü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *ACM Conference on Computer and Communications Security*, pages 213–222, 2009.
- [22] Mohammad Etemad and Alptekin Küpcü. Transparent, distributed, and replicated dynamic provable data possession. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security, ACNS’13*, pages 1–18, Berlin, Heidelberg, 2013. Springer-Verlag.
- [23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP ’03*, pages 29–43, New York, NY, USA, 2003. ACM.
- [24] Jim Gray. Distributed computing economics. *Queue*, 6(3):63–68, May 2008.
- [25] Ari Juels and Burton S. Kaliski Jr. PORs: Proofs Of Retrievability for Large Files. In *ACM Conference on Computer and Communications Security*, pages 584–597, 2007.
- [26] Ghassan Karame and Srdjan Capkun. Low-cost client puzzles based on modular exponentiation. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, pages 679–697, 2010.
- [27] Neal Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.
- [28] Çetin Kara Koç, Tolga Acar, and Burton S. Kaliski, Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
- [29] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, New York, NY, USA, 1986.
- [30] Yadi Ma, Thyaga Nandagopal, Krishna P. N. Puttaswamy, and Suman Banerjee. An ensemble of replication and erasure codes for cloud file systems. In *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013*, pages 1276–1284, 2013.
- [31] NetEm. NetEm, the Linux Foundation. Website, 2009. Available online at <http://www.linuxfoundation.org/collaborate/workgroups/networking/netem>.
- [32] Zachary N. J. Peterson, Mark Gondree, and Robert Beverly. A position paper on data sovereignty: The importance of geolocating data in the cloud. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing, HotCloud’11*, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.
- [33] R. L. Rivest, A. Shamir, and D. A. Wagner. Timelock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, 1996.
- [34] Hovav Shacham and Brent Waters. Compact Proofs of Retrievability. In *ASIACRYPT*, pages 90–107, 2008.
- [35] Marten van Dijk, Ari Juels, Alina Oprea, Ronald L. Rivest, Emil Stefanov, and Nikos Triandopoulos. Hourglass schemes: how to prove that cloud files are encrypted. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 265–280. ACM, 2012.
- [36] Gaven J. Watson, Reihaneh Safavi-Naini, Mohsen Alimomeni, Michael E. Locasto, and Shivaramakrishnan Narayan. LoSt: location based storage. In Ting Yu, Srdjan Capkun, and Seny Kamara, editors, *CCSW*, pages 59–70. ACM, 2012.

A MR-PDP

In what follows, we briefly describe the multi-replica provable data possession scheme by Curtmola *et al.* [18]. Here, the user first splits the file D into

n blocks $d_1, \dots, d_n \in \mathbb{Z}_N$. Let $p = 2p' + 1, q = 2q' + 1$ be safe primes, and $N = pq$ an RSA modulus; moreover, let g be a generator of the quadratic residues of \mathbb{Z}_N^* , and e, d a pair of integers such that $e \cdot d = 1 \pmod{p'q'}$. The user creates authentication tags for each block $i \in [1, n]$ by computing $T_i \leftarrow (h(v||i)g^{d_i})^d \pmod{N}$, where $h: \{0, 1\}^* \rightarrow \mathbb{Z}_N$ is a hash function and $v \in \mathbb{Z}_N$.

Subsequently, each replica is created by the user as follows: $d_i^{(k)} \leftarrow d_i + \text{PRF}(k||i)$ where PRF denotes a pseudorandom function. The user sends to the service provider the tags $\{T_i\}$, the original file blocks $\{d_i\}$, and the replica blocks $d_i^{(k)}$.

At the verification stage, the user selects a replica k and creates a (pseudo-)random challenge set $I = \{(k_1, i_1), \dots, (k_\ell, i_\ell)\}$ where k_j denotes the replica number and i_j the block index. In addition, the user picks $s \in \mathbb{Z}_N^*$, and computes $g_s = g^s \pmod{N}$. The challenge query then comprises the set I and the value g_s which are both sent to the service provider who stores replica k . The service provider then computes the response (T, σ) as follows and sends it back to the verifier:

$$T \leftarrow \prod_{i \in I} T_i, \quad \sigma \leftarrow g_s^{\sum_{1 \leq j \leq \ell} d_{i_j}^{(k_j)}}$$

Finally, the user checks whether:

$$\sigma \stackrel{?}{=} \left(\frac{T^e}{\prod h(v||i)} g^{\sum_{1 \leq j \leq \ell} \text{PRF}(k_j||i_j)} \right)^s$$

B POR Schemes of Shacham and Waters

In what follows, we briefly describe the private POR scheme by Shacham and Waters [34]. This scheme leverages a pseudo-random function PRF. Here, the user first applies an erasure code to the file and then splits it into n blocks $d_1, \dots, d_n \in \mathbb{Z}_p$, where p is a large prime. The user then chooses a random $\alpha \in_R \mathbb{Z}_p$ and creates for each block an authentication value as follows:

$$\sigma_i = \text{PRF}_{key}(i) + \alpha \cdot d_i \in \mathbb{Z}_p. \quad (15)$$

The blocks $\{d_i\}$ and their authentication values $\{\sigma_i\}$ are all stored at the service provider in D^* .

At the POR verification stage, the verifier (here, the user) chooses a random challenge set $I \subset \{1, \dots, n\}$ of size ℓ , and ℓ random coefficients $v_i \in_R \mathbb{Z}_p$. The challenge query then is the set $Q := \{(i, v_i)\}_{i \in I}$ which is sent to the prover (here, service provider). The

prover computes the response (σ, μ) as follows and sends it back to the verifier:

$$\sigma \leftarrow \sum_{(i, v_i) \in Q} v_i \sigma_i, \quad \mu \leftarrow \sum_{(i, v_i) \in Q} v_i d_i.$$

Finally, the verifier checks the correctness of the response:

$$\sigma \stackrel{?}{=} \alpha \mu + \sum_{(i, v_i) \in Q} v_i \cdot \text{PRF}(i).$$

C Improving User Verification in Mirror

In what follows, we describe a number of optimizations that we adopted in our implementation in order to reduce the effort in verifying the service provider's response.

Using either g or h : Recall that the service provider's response involves powers of g and of h which have order p' and q' , respectively. One technique that allows to reduce the effort on the user's side is to rely on either g or h . That is, at the beginning of the verification step, the user randomly decides whether only g or only h shall be taken into account. For example, let us assume that the choice falls on g . Then, the user proceeds as follows:

1. The user computes:

$$\tilde{\sigma} := \sigma^{q'} \cdot \prod_{c=1}^{\ell} \left(\prod_{j=1}^s \prod_{k \in R} g_{i_c, j}^k \right)^{-(q' \cdot v_c)}. \quad (16)$$

Here, we exploit the fact that $(h^e)^{q'} = 1$ for any e .

2. The user checks if:

$$\left(\prod_{j=1}^s \mu_j^{\varepsilon_j + |R|} \right)^{q'} = \tilde{\sigma}. \quad (17)$$

This approach incurs two additional exponentiations but completely eliminates the need to compute the expressions for the values h .

Representing LFSRs by Pre-computed Matrices: According to our experiments, suitable parameter choices are to choose the length λ of the secret LFSR quite small, e.g., equal to 2, while the block size s is comparatively large. This motivates the following optimization.

Let $A_t^{(k)} := (a_t^{(k)}, \dots, a_{t+\lambda-1}^{(k)})$ for any $t \geq 1$ and any $k \in \{1, \dots, r\}$. That is, $A_1^{(k)}$ denotes the initial state of the k -th LFSR while $A_t^{(k)}$ denotes the state after $t - 1$

clocks. Recall that we consider r LFSR sequences which are all generated by the same feedback function. Namely, it holds for any $t \geq 1$ and any $k \in \{1, \dots, r\}$ that $a_{t+\lambda}^{(k)} = \sum_{i=1}^{\lambda} a_i \cdot a_{t+i-1}^{(k)}$. Due to the linearity of the feedback function, there exists a $\lambda \times \lambda$ matrix M , called the companion matrix, for which it holds that:

$$M' \cdot A_1^{(k)} = A_{t+1}^{(k)}, \quad \forall t \geq 0. \quad (18)$$

Recall that we aim to compute for each $i \in I$ the value

$$\sum_{k \in R} \left(a_{\pi(i_c, 1)}^{(k)} + a_{\pi(i_c, 1)+1}^{(k)} + \dots + a_{\pi(i_c, 1)+s-1}^{(k)} \right) \quad (19)$$

where $\pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is a mapping such that $g_{i,j}^{(k)} = g_{\pi(i,j)}^{(k)} = g^{a_j^{(k)}}$ and to raise g by the resulting value. The idea is now to combine as many computations as possible to reduce the overall effort. To this end, the goal is to sum-up for each $k \in R$ and each $i \in I$ the following internal states:

$$\begin{aligned} A_{\pi(i, 1)}^{(k)} &= (a_{\pi(i, 1)}^{(k)}, \dots, a_{\pi(i, 1)+\lambda-1}^{(k)}) \\ &\vdots \\ A_{\pi(i, 1)+\lfloor s/\lambda \rfloor \cdot \lambda}^{(k)} &= (a_{\pi(i, 1)+\lfloor s/\lambda \rfloor \cdot \lambda}^{(k)}, \dots, a_{\pi(i, 1)+\lfloor s/\lambda \rfloor \cdot \lambda-1}^{(k)}) \end{aligned}$$

This can be accomplished by computing:

$$\left(\sum_{i \in I} M^{\pi(i, 1)} \right) \cdot \left(\sum_{j=0}^{\lfloor s/\lambda \rfloor} M^{j \cdot \lambda} \right) \cdot \left(\sum_{k \in R} A_1^{(k)} \right). \quad (20)$$

Observe that $\sum_{j=0}^{\lfloor s/\lambda \rfloor} M^{j \cdot \lambda}$ is independent of the current challenge and can be precomputed. Moreover, due to the fact that we aim for a small LFSR length, e.g., $\lambda = 2$, the user may consider to precompute the value in the last bracket for any choice of R , yielding an additional storage effort of $\lambda \cdot (2^r - 1)$ sectors. In such case, the computation would boil down to the effort of computing the first bracket only. We note that if λ does not divide s , then the user has to compute in addition:

$$\left(\sum_{i \in I} M^{\pi(i, 1)} \right) \cdot M^{\lceil s/\lambda \rceil \cdot \lambda} \cdot \left(\sum_{k \in R} A_1^{(k)} \right), \quad (21)$$

and to add the sum of the first $s \bmod \lambda$ entries to the value computed above. Also here, similar precomputations can be done to accelerate this step.

D Valid Relations

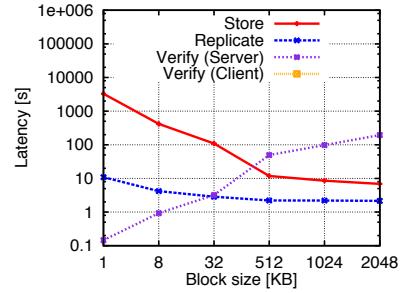
We now explain why $\vec{v} = (v_1, \dots, v_{n \cdot s}) \in \mathbb{Z}^{n \cdot s}$ such that

$$\prod_{i=1}^{n \cdot s} g_i^{v_i} = 1, \quad (22)$$

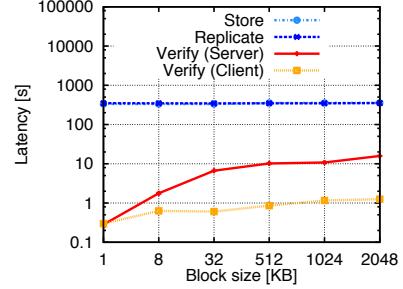
represents the only type of equations that allows the provider to compute missing values g_j from known values g_i .

To see why, recall that $g_j = g^{a_j}$ where $(a_j)_j$ represents an LFSR-sequence. More precisely, this sequence is defined by the feedback polynomial and the initial state, i.e., the first λ entries. Without knowing these entries, it is (information-theoretically) impossible to determine a_j for larger indices. Also, any element in $(a_j)_j$ is a linear combination of the initial state values. Let us denote this combination by L_1 . This can be relaxed as follows: the knowledge of any λ elements $a_{j_1}, \dots, a_{j_\lambda}$ of the sequence $(a_j)_j$ allows almost always to compute another element by an appropriate linear combination (say L_2) of $a_{j_1}, \dots, a_{j_\lambda}$. Notice that the coefficients of L_2 have to be a linear combination of the coefficients (or shifted versions) of L_1 (this is an inherent property of LFSRs). This is exactly the definition of “valid relations” given in Equation (12).

E Impact of the Block Size on the Performance of MR-PDP and Mirror



(a) Impact of block size on the performance of MR-PDP [18].



(b) Impact of block size on the performance of Mirror.

Figure 4: Impact of the block size on the performance of MR-PDP [18] and Mirror.

ZKBoo: Faster Zero-Knowledge for Boolean Circuits

Irene Giacomelli Jesper Madsen Claudio Orlandi
Computer Science Department, Aarhus University

Abstract

In this paper we describe ZKBoo¹, a proposal for practically efficient zero-knowledge arguments especially tailored for Boolean circuits and report on a proof-of-concept implementation. As a highlight, we can generate (resp. verify) a non-interactive proof for the SHA-1 circuit in approximately 13ms (resp. 5ms), with a proof size of 444KB.

Our techniques are based on the “MPC-in-the-head” approach to zero-knowledge of Ishai et al. (IKOS), which has been successfully used to achieve significant asymptotic improvements. Our contributions include:

- A thorough analysis of the different variants of IKOS, which highlights their pros and cons for practically relevant soundness parameters;
- A generalization and simplification of their approach, which leads to faster Σ -protocols (that can be made non-interactive using the Fiat-Shamir heuristic) for statements of the form “I know x such that $y = \phi(x)$ ” (where ϕ is a circuit and y a public value);
- A case study, where we provide explicit protocols, implementations and benchmarking of zero-knowledge protocols for the SHA-1 and SHA-256 circuits.

1 Introduction

Since their introduction in the 80s [16], zero-knowledge (ZK) arguments have been one of the main building blocks in the design of complex cryptographic protocols. However, due to the lack of practically efficient solutions for proving generic statements, their application in real-world systems is very limited. In particular, while there is a large body of work considering the efficiency of ZK protocols for algebraic languages (following the seminal work of Schnorr for discrete logarithm [26]), things are quite different when it comes to general purpose ZK.

A notable exception is the recent line of work on *succinct non-interactive arguments of knowledge* (SNARKs) (e.g. Pinocchio [23], libsnark [4], etc.). SNARKs are an extremely useful tool when the size of the proof and the verification time matters: SNARKs are less than 300 bytes and can be verified in the order of 5ms, which makes them perfect for applications such as ZeroCash [3]. However, on the negative side, SNARKs require very large parameters (which must be generated in a trusted way) and the time to generate proofs are prohibitive for many applications. As an example, the running time of the prover for generating a proof for SHA-1 is in the order of 10 seconds. There is an inherent reason for this inefficiency: current SNARKs technology requires to perform expensive operations (in pairing friendly groups) for each gate in the circuit.

Jawurek et al. [21] proposed a different approach to efficient ZK, namely using garbled circuits (GC). Using GC, it is possible to prove any statement (expressed as a Boolean circuit) using only a (low) constant number of symmetric key operations per gate in the circuit, thus decreasing the proving time by more than an order of magnitude. On the flip-side, GC-based ZK are inherently interactive, and they still require a few public-key operations (used for implementing the necessary oblivious transfers).

In this paper we describe efficient ZK protocols for circuits based on the “MPC-in-the-head” paradigm of Ishai et al. [19] (IKOS). In IKOS, a prover simulates an MPC protocol between a number of “virtual” servers (at least 3) and then commits to the views and internal state of the individual servers. Now the verifier challenges the prover by asking to open a subset of these commitments. The privacy guarantee of the underlying MPC protocol guarantees that observing the state of a (sufficiently small) subset of servers does not reveal any information. At the same time, the correctness of the MPC protocol guarantees that if the prover tries to prove a false statement, then the joint views of some of the server must nec-

¹Sounds like Peekaboo.

essarily be inconsistent, and the verifier can efficiently check that. By plugging different MPC protocols into this approach, [19] shows how to construct ZK protocols with good asymptotic properties. However, to the best of our knowledge, no one has yet investigated whether the IKOS approach can be used to construct practically efficient ZK protocols. This paper is a first step in this direction.

Structure of the paper. In Section 3 we describe the different variants of the IKOS framework. IKOS presents two strategies to achieve a negligible soundness error: either repeating a passive secure MPC protocol with few parties, or using a single instance of an active secure MPC protocol with a large number of parties. While IKOS only provides asymptotic estimates of the soundness parameters, we concretely estimate the soundness of IKOS with different kind of MPC protocols and show that, if one is interested in a (reasonable) soundness error of 2^{-80} , then the version of IKOS without repetition does not (unfortunately) lead to any practical advantage. Then (in Section 4) we present a new interpretation of the IKOS framework when instantiated with a 2-private 3-party version of the GMW [15] protocol, where each pair of parties is connected with an OT-channel. We observe that in general the OT-channels can be replaced with arbitrary 2-party functionalities. Since those ideal functionalities do not have to be implemented using cryptographic protocols (remember, they are executed between pair of virtual servers in a simulation performed by the prover), this increases the degrees of freedom of the protocol designer and allows to construct more efficient MPC protocols (or, as we prefer to call them, function decompositions) that can be used for constructing ZK protocols. (Note that this class of protocol has not been studied before, since it does not lead to any advantage in the standard MPC setting, and therefore we expect future work to improve on our approach by designing better MPC protocols for this special setting.) All resulting protocols are Σ -protocols (3-move honest-verifier zero-knowledge protocols with special soundness) which can therefore be made *non-interactive* in the random oracle model using the Fiat-Shamir heuristic.

Finally (in Section 5) we describe how our approach can be used to construct very efficient ZK protocols for proving knowledge of preimages for SHA-1 and SHA-256. The resulting proofs are incredibly efficient: the verification time is essentially the same as the verification time for SNARKs, but the prover runs approximately 1000 times faster. On the negative side the size of our proofs scales linearly with the circuit size, but we believe that in some applications this is a desirable trade-off.

Recent Related Work. Ranellucci *et al.* [25] proposed a general-purpose public-coin ZK protocol which can be based on any commitment scheme. The asymptotic performances are the same as ours (both communication and computation complexity are linear in the circuit size) but the concrete constants are higher (e.g., the proofs are approximately 3 times larger and computation more than 10 times slower). Hazay *et al.* [18] show how to extend the IKOS technique to the case of two-party MPC protocols (2PC) with application to *adaptive* ZK protocols. It is an open question whether their approach might lead to concrete efficiency improvements.

2 Preliminaries

Standard notations: For an integer n , we write $[n] = \{1, 2, \dots, n\}$ and, given $A \subseteq [n]$, $|A|$ denotes the cardinality of A . We say that a function ε is *negligible* in n , $\varepsilon(n) = negl(n)$, if for every polynomial p there exists a constant c such that $\varepsilon(n) < \frac{1}{p(n)}$ when $n > c$. Given two random variables X ad Y with support S , the statistical distance between X and Y is defined as $SD(X, Y) = \frac{1}{2} \sum_{i \in S} |\Pr[X = i] - \Pr[Y = i]|$. Two families $X = \{X_k\}$ and $Y = \{Y_k\}$, $k \in \{0, 1\}^*$ of random variables are said to be *statistically indistinguishable* if there exists a negligible function $\varepsilon(\cdot)$ such that for every $k \in \{0, 1\}^*$, $SD(X_k, Y_k) \leq \varepsilon(|k|)$. They are said to be *computationally indistinguishable* if for every efficient non-uniform distinguisher D there exists a negligible function $\varepsilon(\cdot)$ such that for every $k \in \{0, 1\}^*$, $|\Pr[D(X_k) = 1] - \Pr[D(Y_k) = 1]| \leq \varepsilon(|k|)$.

2.1 Multi-Party Computation (MPC)

Consider a public function $f : (\{0, 1\}^k)^n \rightarrow \{0, 1\}^\ell$ and let P_1, \dots, P_n be n players modelled as PPT machines. Each player P_i holds the value $x_i \in \{0, 1\}^k$ and wants to compute the value $y = f(x)$ with $x = (x_1, \dots, x_n)$ while keeping his input private. The players can communicate among them using point-to-point secure channels $CH_{i,j}$ in the synchronous model. These can be classical secure channels (*i.e.* encrypted channels) or more powerful channels (*e.g.* OT-channel [11, 24]). If necessary, we also allow the players to use a broadcast channel. To achieve their goal, the players jointly run a n -party MPC protocol Π_f . The latter is a protocol for n players that is specified via the next-message functions: there are several rounds of communication and in each round the player P_i sends into the channel $CH_{i,j}$ (or in the broadcast channel) a message that is computed as a deterministic function of the internal state of P_i (his initial input x_i and his random tape k_i) and the messages that P_i has received in the previous rounds of communications. The view of the player P_j , denoted by $View_{P_j}(x)$, is defined as the

concatenation of the private input \mathbf{x}_j , the random tape \mathbf{k}_j and all the messages received by P_j during the execution of Π_f . Each channel $\text{CH}_{i,j}$ defines a relation of *consistency* between views. For instance, in a plain channel *two views are consistent if* the messages reported in $\text{View}_{P_j}(\mathbf{x})$ as incoming from P_i are equal to the outgoing message implied by $\text{View}_{P_i}(\mathbf{x})$ ($i \neq j$). More powerful channels (such as OT channels), are defined via some function φ and we say that *two views are consistent if* the view of the sender implies an input \mathbf{x} to the channel and the view of the receiver implies an input \mathbf{y} and contains an output \mathbf{z} such that $\mathbf{z} = \varphi(\mathbf{x}, \mathbf{y})$. For instance, in OT channels $\mathbf{x} = (m_0, m_1)$, \mathbf{y} is a bit and $\mathbf{z} = m_y$.

Finally, the output \mathbf{y} can be computed from any of the view $\text{View}_{P_i}(\mathbf{x})$, *i.e.* there are n functions $\Pi_{f,1}, \dots, \Pi_{f,n}$ such that $\mathbf{y} = \Pi_{f,i}(\text{View}_{P_i}(\mathbf{x}))$ for all $i \in [n]$. In order to be private, the protocol Π_f needs to be designed in such a way that a curious player P_i can not infer information about \mathbf{x}_j with $j \neq i$ from his view $\text{View}_{P_i}(\mathbf{x})$. An additional security property, robustness, assures that a cheating player P_i (who may not follow the instructions in the protocol) can not mislead the honest players, who still compute the correct output \mathbf{y} . More precisely, we have the following definition.

Definition 2.1. • (*Correctness*) We say that the protocol Π_f realizes f with perfect (resp. statistical) correctness if for any input $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$, it holds that $\Pr[f(\mathbf{x}) \neq \Pi_{f,i}(\text{View}_{P_i}(\mathbf{x}))] = 0$ (resp. negligible) for all $i \in [n]$. The probability is over the choice of the random tapes \mathbf{k}_i .

• (*Privacy*) Let $1 \leq t < n$, the protocol Π_f has perfect t -privacy if it is correct and for all $A \subseteq [n]$ satisfying $|A| \leq t$ there exists a PPT algorithm S_A such that the joint views $(\text{View}_{P_i}(\mathbf{x}))_{i \in A}$ have the same distribution as $S_A(f, (\mathbf{x}_i)_{i \in A}, \mathbf{y})$, for all $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$.

We will speak about statistical (resp. computational) t -privacy if the two distributions $S_A(f, (\mathbf{x}_i)_{i \in A}, \mathbf{y})$ and $(\text{View}_{P_i}(\mathbf{x}))_{i \in A}$ are statistically (resp. computationally) indistinguishable.

• (*Robustness*) Let $0 \leq r < n$, the protocol Π_f has perfect (resp. statistical) r -robustness if it is correct and for all $A \subseteq [n]$ satisfying $|A| \leq r$ even assuming that all the players in A have been arbitrarily corrupted, then $\Pr[f(\mathbf{x}) \neq \Pi_{f,i}(\text{View}_{P_i}(\mathbf{x}))] = 0$ (resp. negligible) for all $i \in A^c$.

3 Zero Knowledge

In this section we recall the notion of zero-knowledge and Σ -protocols, we review the IKOS construction [19] for zero-knowledge, and we discuss different possible instantiations.

3.1 Definitions

Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation representing some computational problem (*e.g.* $R = \{(\mathbf{y}, \mathbf{x}) \mid \mathbf{y} = \text{SHA-256}(\mathbf{x})\}$). We will interpret R as a binary function from $\{0,1\}^* \times \{0,1\}^*$ to $\{0,1\}$ (*i.e.* $R(\mathbf{y}, \mathbf{x}) = 1 \Leftrightarrow (\mathbf{y}, \mathbf{x}) \in R$) and we will assume that:

- $\forall \mathbf{y}$ and $\forall \mathbf{x}$, $R(\mathbf{y}, \mathbf{x})$ can be computed in polynomial-time by a probabilistic Turing machine;
- there exists a polynomial p such that if $R(\mathbf{y}, \mathbf{x}) = 1$ then the length of \mathbf{x} is less or equal to $p(|\mathbf{y}|)$.

Such relation is called NP relation. With L we indicate the set of the yes-instances of the relation R , *i.e.* $L = \{\mathbf{y} \mid \exists \mathbf{x} \text{ s.t. } R(\mathbf{y}, \mathbf{x}) = 1\}$.

An *argument* for L is a cryptographic protocols between two players: the prover P and the verifier V with the following features. We assume that both P and V are probabilistic polynomial time (PPT) machines and that they know \mathbf{y} , an instance of the relation R . The situation is that P wants to convince V that $\mathbf{y} \in L$. This clearly makes sense only if the prover has some advantage over the verifier. Thus, we allow the prover to have an extra private input (for example P knows \mathbf{x} such that $R(\mathbf{y}, \mathbf{x}) = 1$). The protocol is described by instructions for the players and has different rounds of communication. At the end of the protocol, the verifier outputs accept if he is convinced or reject otherwise. If $\mathbf{y} \in L$, we require that an honest verifier convinces an honest prover with probability 1 (the protocol is complete). On the other hand, we say that the protocol has *soundness error* ϵ if for all $\mathbf{y} \notin L$ $\Pr[V(\mathbf{y}) = \text{accept}] \leq \epsilon$, no matter what the prover does. In other words, ϵ is an upper-bound of the probability that a cheating prover makes an honest verifier output accept for a false instance.

However, in many interesting cryptographic applications, the language L is trivial and therefore the soundness property gives absolutely no guarantees: for every string \mathbf{y} there exist a \mathbf{x} s.t., $\mathbf{y} = \text{SHA-256}(\mathbf{x})$. In this case we need a stronger property, namely *proof-of-knowledge* (PoK), which informally states that the verifier should output accept only if the prover *knows* the value \mathbf{x} .

Finally, ZK protocols get their name from the *zero-knowledge* property: Here, we want to express the requirement that whatever strategy a cheating verifier follows, he learns nothing except for the truth of the prover's claim. In particular, he can not obtain information about the private input of P . This is captured using the simulation-paradigm and saying that the messages received by the verifier during the protocol can be efficiently simulated only knowing the public input \mathbf{y} . More precisely, we have the following requirement: for any corrupted PPT verifier V^* , there is a PPT algorithm S (the “simulator”) with access to V^* such that the output

of $S(\mathbf{y})$ and the real conversation between P and V^* on input \mathbf{y} are indistinguishable.

In the rest of the paper we will be concerned with public-coin two-party protocols with a specific communication pattern known as Σ -protocols.

Definition 3.1 (Σ -protocol). *A protocol Π_R between two players P and V is a Sigma Protocol for the relation R if it satisfies the following conditions:*

- Π_R has the following communication pattern:
 1. (Commit) P sends a first message \mathbf{a} to V ;
 2. (Challenge) V sends a random element \mathbf{e} to P ;
 3. (Prove) P replies with a second message \mathbf{z} .
- (**Completeness**) If both players P and V are honest and $\mathbf{y} \in L$, then $\Pr[(P,V)(\mathbf{y}) = \text{accept}] = 1$;
- (**s -special soundness**) For any \mathbf{y} and any set of s accepting conversations $\{(\mathbf{a}, \mathbf{e}_i, \mathbf{z}_i)\}_{i \in [s]}$ with $\mathbf{e}_i \neq \mathbf{e}_j$ if $i \neq j$, a witness \mathbf{x} for \mathbf{y} can be efficiently computed;
- (**Special honest-verifier ZK**) There exists a PPT simulator S such that on input $\mathbf{y} \in L$ and \mathbf{e} outputs a triple $(\mathbf{a}', \mathbf{e}', \mathbf{z}')$ with same probability distribution of real conversations $(\mathbf{a}, \mathbf{e}, \mathbf{z})$ of the protocol.



Figure 1: The communication pattern of a Σ -protocol.

Σ -protocols have several properties (e.g. parallel composition, witness indistinguishability) that make them a useful building block for many other cryptographic primitives (identification schemes, signatures, etc). See [7] or [17, Chapter 6] for more details on this. Here we are mainly interested in the following facts: First, Σ -protocols are public-coin protocols and thus they can be made non-interactive in the random oracle model using the Fiat-Shamir heuristic [12]. Second, there exist efficient transformations from Σ -protocols to fully-fledged ZK and PoK: indeed, it is possible to efficiently transform a Σ -protocol into a zero-knowledge argument (resp. zero-knowledge proof of knowledge) with the addition of one additional round (resp. two additional rounds). Note

finally that if the challenge \mathbf{e} is chosen uniformly at random form a set of cardinality c , then s -special soundness implies a bound of $(s-1)/c$ on the soundness error of the protocol: if $\mathbf{y} \notin L$, then there exist no \mathbf{x} s.t. $R(\mathbf{x}, \mathbf{y}) = 1$, and therefore fixed any \mathbf{a} there are at most $s-1$ challenges such that an accepting conversation for them exists.

3.2 IKOS Construction

In 2007 Ishai *et al.* show how to use any MPC protocol and the commitment-hybrid (Com) model² to obtain a ZK proof for an arbitrary NP relation R with asymptotically small soundness error. Here we briefly recall their construction and moreover we explicitly analyse its soundness error.

Let Π_f be an MPC protocol that realizes any n -party function f with perfect correctness (Definition 2.1). Depending on the features of Π_f (privacy, robustness, communication channels used), [19] presents slightly different ZK protocols. However, the general structure is always the same and is the structure of a Σ -protocol, see Figure 1. The high-level idea is the following: assume that $\mathbf{y} \in L$ is the public input of the ZK protocol, while \mathbf{x} is the private input of the prover (*i.e.* $R(\mathbf{y}, \mathbf{x}) = 1$). The prover first takes n random values $\mathbf{x}_1, \dots, \mathbf{x}_n$ such that $\mathbf{x} = \mathbf{x}_1 \oplus \dots \oplus \mathbf{x}_n$, then he considers the n -input function f_y defined as

$$f_y(\mathbf{x}_1, \dots, \mathbf{x}_n) := R(\mathbf{y}, \mathbf{x}_1 \oplus \dots \oplus \mathbf{x}_n)$$

and emulates “in his head” the protocol Π_{f_y} on inputs $\mathbf{x}_1, \dots, \mathbf{x}_n$. After the emulation, he computes the commitments to each of the n produced views (*i.e.* $\text{Com}(\text{View}_{P_i}(\mathbf{x}))$ for $i = 1, \dots, n$). After all the commitments have been stored, the verifier challenges the prover to open some of them (*i.e.* the challenge is a random subset of $[n]$ of a given size). Finally, the prover opens the requested commitments and the verifier outputs accept if and only if all the opened views are consistent with each other and with output 1.

Here we focus on the ZK protocols presented in [19] that assume a perfectly correct (and eventually perfectly robust) MPC protocol and we collect them in two versions. Version 1 considers the case of an MPC protocol with t -privacy and perfect r -robustness with $t > 1$.³ Version 2 shows that 2-privacy is not necessary condition and indeed considers the case of an MPC protocol with 1-privacy only.

²In the *commitment-hybrid model* the two parties have access to an idealized implementation of commitments, which can be imagined as a trusted third party which stores the messages of the sender and only reveals them if told so by the sender.

³This is a generalization of [19] as they only consider the case $t = r$.

IKOS Protocol (Version 1)

The verifier and the prover have input $\mathbf{y} \in L$. The prover knows \mathbf{x} such that $R(\mathbf{y}, \mathbf{x}) = 1$. A perfectly correct and t -private n -party MPC protocol Π_{f_y} is given ($2 \leq t < n$).

Commit: The prover does the following:

1. Sample random vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ s.t. $\mathbf{x}_1 \oplus \dots \oplus \mathbf{x}_n = \mathbf{x}$;
2. Run $\Pi_{f_y}(\mathbf{x}_1, \dots, \mathbf{x}_n)$ and obtain the views $\mathbf{w}_i = \text{View}_{P_i}(\mathbf{x})$ for all $i \in [n]$;
3. Commit to $\mathbf{w}_1, \dots, \mathbf{w}_n$.

Prove: The verifier chooses a subset $E \subseteq [n]$ such that $|E| = t$ and sends it to the prover. The prover reveals the value \mathbf{w}_e for all $e \in E$.

Verify: The verifier runs the following checks:

1. If $\exists e \in E$ s.t. $\Pi_{f,e}(\text{View}_{P_e}(\mathbf{x})) \neq 1$, output reject;
2. If $\exists \{i, j\} \subset E$ s.t. $\text{View}_{P_i}(\mathbf{x})$ is not consistent with $\text{View}_{P_j}(\mathbf{x})$, output reject;
3. Output accept;

Figure 2: The IKOS zero-knowledge protocol for the relation R in the commitment-hybrid model.

Version 1: Let t and r be two integers, $2 \leq t < n$ and $0 \leq r \leq t$. We assume that the protocol Π_{f_y} is perfectly correct and satisfies two more properties: perfect, statistical or computational t -privacy and perfect r -robustness. In this version of the IKOS protocol (Figure 2) the verifier is allowed to ask for the openings of t of the commitments $\text{Com}(\text{View}_{P_i}(\mathbf{x}))$. In this way, the zero-knowledge property follows easily by the t -privacy of the protocol Π_{f_y} .

For the analysis of the soundness error of this protocol we use the so-called *inconsistency graph* G . Given an execution of Π_{f_y} , the graph G has n nodes and there is an edge (i, j) if and only if the views of the players P_i and P_j are inconsistent. Assume that $\mathbf{y} \notin L$ and that the execution of Π_{f_y} is not a correct one (otherwise $\Pr[V(\mathbf{y}) = \text{accept}] = 0$ because of the checks in step 1 of the procedure **Verify**). Then we have two cases:

1. There is in G a vertex cover set⁴ B of size at most

⁴ B is a vertex cover set for the graph G if each edge in G is incident to at least 1 node in B .

⁵ $\binom{r}{t}$ is 0 if $r < t$.
⁶A matching is a set of edges without common nodes.

r . Intuitively, this means that in the current execution of Π_{f_y} only the players in B have been actively corrupted. Indeed, if we remove the nodes in B , we obtain a graph without edges. That is, all the players not in B have views consistent among them and we can consider these players honest. Since the size of B is less or equal to the parameter r , the robustness property assures that for all the players not in B (honest players) the view implies a 0 output (the correct output of the protocol Π_{f_y}). The probability that the verifier will not see one of these views choosing t of them uniformly at random is less or equal to⁵

$$p_1(n, t, r) = \binom{r}{t} \binom{n}{t}^{-1}$$

2. If the size of the minimum vertex cover is $> r$, then the graph G has a matching⁶ of size $> r/2$. The probability that the verifier accepts the wrong proof is equal to the probability that between the t nodes that he chooses there are no edges of G and this is less or equal to the probability that there are no edges from the matching. Clearly, this probability reaches the maximum when the matching is the smallest possible, that is it has size $k = \lfloor r/2 \rfloor + 1$. In this situation the aforementioned probability is

$$p_2(n, t, r) = \begin{cases} 0 & \text{otherwise} \\ \left(\sum_{j=0}^k 2^j \binom{k}{j} \binom{n-2k}{t-j} \right) \binom{n}{t}^{-1} & \text{if } n - 2k > 0 \end{cases}$$

In general, the soundness error is equal to the value $p(n, t, r) = \max\{p_1(n, t, r), p_2(n, t, r)\}$.

Version 2: A second version of the protocol was proposed in [20] to show that 2-privacy is not a necessary condition for the IKOS construction. In other words, we can construct ZK proofs in from 1-private MPC protocols. Notice that in this case the MPC protocol is allowed to use only standard point-to-point secure channels. The idea of the construction is very similar to the previous one, but now the prover commits to all the $\binom{n}{2}$ channels in addition to committing to the n views. The verifier picks a random $i \in [n]$ and challenges the prover to open the view of the player P_i and all the $n - 1$ channels CH_{ij} incident to him. Finally, the verifier accepts if the opened view is consistent with the channels and with the output 1. Again, the ZK property follows from the privacy property of the MPC protocol: the information revealed to the verifier is implied by the view of a single player. To compute the soundness error in this case, observe that for any

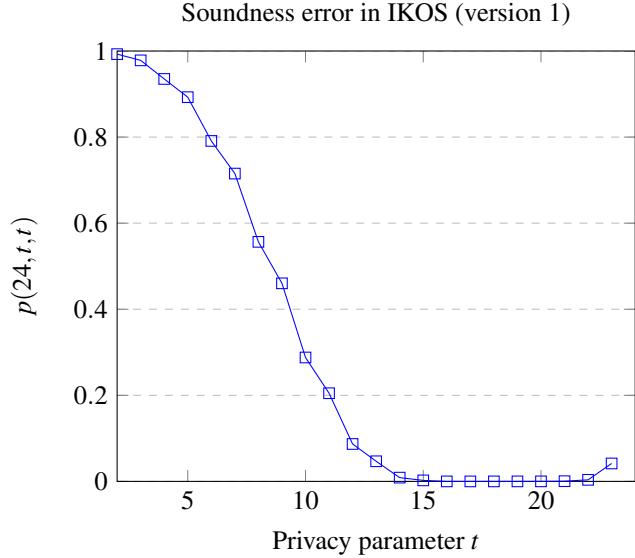


Figure 3: The graph represents the soundness error $p(n, t, r)$ in function of t when $t = r$ and $n = 24$. The table shows the values of σ such that $p(24, t, t) = 2^{-\sigma}$ for $t \in \{15, \dots, 20\}$.

incorrect execution of Π_{f_y} there is at least one player P_i such that $\text{View}_{P_i}(\mathbf{x})$ is inconsistent with a channel CH_{ij} . The probability $\Pr[V(\mathbf{y}) = \text{accept}]$ is less or equal to the probability that V does not choose this index i . Therefore, the soundness error of this version is $1 - 1/n$.

3.3 Our choice of version and parameters

In this section we discuss and motivate some of our design choices.

Which MPC protocol? As discussed, IKOS can be instantiated with a large number of MPC protocols. In particular, using MPC protocols with good asymptotic properties (such as [8, 9], etc.), one can obtain ZK protocols with equally good asymptotic properties. However in this paper we are concerned with concrete, constant size circuits, and we do not want to put any restriction on the shape or width of the circuits. Thus, the best two choices are BGW [2] style protocols with $t = r = \lfloor \frac{n-1}{3} \rfloor$ which use simple point-to-point channels and GMW [15] style protocols with $t = n-1, r = 0$ which use OT channels between each pair of parties. Then we have the following two cases:

1. (GMW [15]:) In this case the soundness error is $\frac{2}{n}$

and we open $n-1$ views. Note that in these protocols each party must communicate with every other party, thus the size of the proof for soundness $2^{-\sigma}$ is given by

$$c \cdot \frac{(n-1)^2}{\log_2(n)-1} \cdot \sigma$$

where c is a constant which depends on the exact protocol. It is easy to see that the function grows with n and therefore smallest proofs are achieved with $n = 3$. Looking ahead, our protocol in Section 4 has $c = 1/2$ and $\sigma = 80$ and therefore the size of the proof is 274 bits per multiplication gate.

2. (BGW [2]:) In this case the soundness error is given by $p_2(n, \lfloor \frac{n-1}{3} \rfloor, \lfloor \frac{n-1}{3} \rfloor)$. To get soundness error $\leq 2^{-80}$, we get that $n \geq 1122$ and therefore the number of opened views is $\lfloor \frac{n-1}{3} \rfloor = 373$. Thus, even if each party only had to store a single bit for each multiplication gate, the size of the proof would already be larger than in the previous case.
3. (Future Work:) Our analysis shows that using an MPC protocol with $t = r = \lceil \frac{2}{3}n \rceil$ it would be enough to use $(n, t, r) = (92, 64, 64)$ to achieve soundness 2^{-80} . The existence of such a protocol, where in addition each party only needs to store ≤ 4 bits per multiplication gate, would give rise to ZK proofs of size smaller than the one we construct. We are not aware of any such protocols, however we cannot rule out their existence. In particular, we note that such protocols have *not* been considered in the literature, since they give rise to poor MPC protocols in practice (note that such a protocol necessarily uses advanced channels, which in the standard MPC protocol need to be implemented using expensive cryptographic operations), and we believe that the quest for “MPC protocols” optimized for the ZK applications has just begun. Figure 3 shows how, for a fixed number of parties n , the soundness error decreases as a function of $t = r$. Note that the soundness error for $\frac{2}{3}n$ is much smaller than $\frac{1}{3}n$.

Why only perfect correctness and robustness? [19] presented also two extensions of the basic construction that allow to use MPC protocol with statistical correctness or with statistical robustness, but we are not considering those cases here for two reasons: first, the resulting ZK protocols have higher round complexity (and are therefore not Σ -protocols); second, perfectly secure MPC protocols are *more efficient*: practically efficient MPC protocols which only achieve statistical security (even when allowing arbitrary two-party channels, such as in [5, 22]) require parties to store *tags* or *MACs* together with their shares, and to make sure that the sta-

tistical error is negligibly small these tags need to be at least as long as the security parameter⁷, whereas in perfectly secure MPC protocols the share size can be made constant.

Why not Version 2? Note that the soundness error of Version 1 with $(n, t, r) = (3, 2, 0)$ is the same as the soundness error of Version 2 with $(n, t, r) = (3, 1, 0)$, thus the number of required rounds is exactly the same. However (i) Version 2 requires to compute and open more commitments and (ii) Version 2 only works with plain channels, while Version 1 allows to use arbitrary channels which helps in constructing more efficient protocols.

4 Generalizing IKOS

This section contains a generalized and optimized version of the IKOS protocol that works for any relation defined by a function, $\phi : X \rightarrow Y$ which can be decomposed in the “right way”. In particular, in Section 4.2 we will describe a ZK Σ -protocol for the relation R_ϕ defined by $R_\phi(\mathbf{y}, \mathbf{x}) = 1 \Leftrightarrow \phi(\mathbf{x}) = \mathbf{y}$, while the decomposition used to construct it is formalized in the following section.

Protocol Π_ϕ^*

Let $\phi : X \rightarrow Y$ be a function and \mathcal{D} a related $(2, 3)$ -decomposition as defined in Definition 4.1.

Input: $\mathbf{x} \in X$

1. Sample random tapes $\mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3$;
2. Compute $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) \leftarrow \text{Share}(\mathbf{x}; \mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3)$;
3. Let $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$ be vectors with $N + 1$ entries;
 - Initialize $\mathbf{w}_i[0] = \mathbf{x}_i$ for all $i \in \{1, 2, 3\}$;
 - For $j = 1, \dots, N$, compute:
 - For $i = 1, 2, 3$, compute
$$\mathbf{w}_i[j] = \phi_i^{(j)}((\mathbf{w}_m[0..j-1], \mathbf{k}_m)_{m \in \{i, i+1\}})$$
4. Compute $\mathbf{y}_i = \text{Output}_i(\mathbf{w}_i, \mathbf{k}_i)$ for $i \in \{1, 2, 3\}$;
5. Compute $\mathbf{y} = \text{Rec}(\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3)$;

Output: $\mathbf{y} \in Y$

Figure 4: Given a correct decomposition \mathcal{D} , the protocol Π_ϕ^* can be used to evaluate the function ϕ .

⁷This can be avoided for SIMD computations [10].

4.1 $(2, 3)$ -Function Decomposition

Given an arbitrary function $\phi : X \rightarrow Y$ and an input value $\mathbf{x} \in X$ we want to compute the value $\phi(\mathbf{x})$ splitting the computation in 3 branches such that the values computed in 2 branches reveals no information about the input \mathbf{x} . In order to achieve this, we start by “splitting” the value \mathbf{x} in three values $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ (called *input shares*) using a surjective function that we indicate with *Share*. These input shares as well as all the intermediate values are stored in 3 string $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$ called the *views*. More precisely, \mathbf{w}_i contains the values computed in the computation branch i . In order to achieve the goal and compute the value $\mathbf{y} = \phi(\mathbf{x})$, we use a finite family of efficiently computable functions that we indicate with $\mathcal{F} = \bigcup_{j=1}^N \{\phi_1^{(j)}, \phi_2^{(j)}, \phi_3^{(j)}\}$. The function $\phi_m^{(j)}$ takes as inputs specific values from the views $\mathbf{w}_m, \mathbf{w}_{m+1}$ with $m = \{1, 2, 3\}$ and where $3 + 1 = 1$. The functions are used in the following way: we use functions $\phi_1^{(j)}, \phi_2^{(j)}, \phi_3^{(j)}$ to compute the next value to be stored in each view \mathbf{w}_m : The function $\phi_m^{(1)}$ takes as input $\mathbf{w}_m, \mathbf{w}_{m+1}$ (which at this point contain only the shares $\mathbf{x}_m, \mathbf{x}_{m+1}$) and outputs one value which is saved in position 1 of the views \mathbf{w}_m . We continue like this for all N functions, with the difference that in step $j > 1$, the function $\phi_m^{(j)}$ can receive as input (any subset of) the current views $\mathbf{w}_m, \mathbf{w}_{m+1}$. The initial function *Share* and all subfunctions $\phi_m^{(j)}$ are allowed to be randomized, and they get their coins from $\mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3$, three random tapes which correspond to the three branches. Finally, after the N steps described, the 3 functions $\text{Output}_1, \text{Output}_2, \text{Output}_3$ are used to compute the values $\mathbf{y}_i = \text{Output}_i(\mathbf{w}_i)$ that we call *output shares*. From these three values we compute the final output $\mathbf{y} = \phi(\mathbf{x})$ using the function *Rec*. The entire procedure is described in detail in Figure 4 (Protocol Π_ϕ^*).

Definition 4.1. A $(2, 3)$ -decomposition for the function ϕ is the set of functions

$$\mathcal{D} = \{\text{Share}, \text{Output}_1, \text{Output}_2, \text{Output}_3, \text{Rec}\} \cup \mathcal{F}$$

such that *Share* is a surjective function and $\phi_m^{(j)}$, Output_i and *Rec* are functions as described before. Let Π_ϕ^* be the algorithm described in Figure 4, we have the following definitions.

- (**Correctness**) We say that \mathcal{D} is correct if $\Pr[\phi(\mathbf{x}) = \Pi_\phi^*(\mathbf{x})] = 1$ for all $\mathbf{x} \in X$. The probability is over the choice of the random tapes \mathbf{k}_i .
- (**Privacy**) We say that \mathcal{D} has 2-privacy if it is correct and for all $e \in [3]$ there exists a PPT simulator S_e such that

$$(\{\mathbf{k}_i, \mathbf{w}_i\}_{i \in \{e, e+1\}}, \mathbf{y}_{e+2}) \text{ and } S_e(\phi, \mathbf{y})$$

have the same probability distribution for all $\mathbf{x} \in X$.

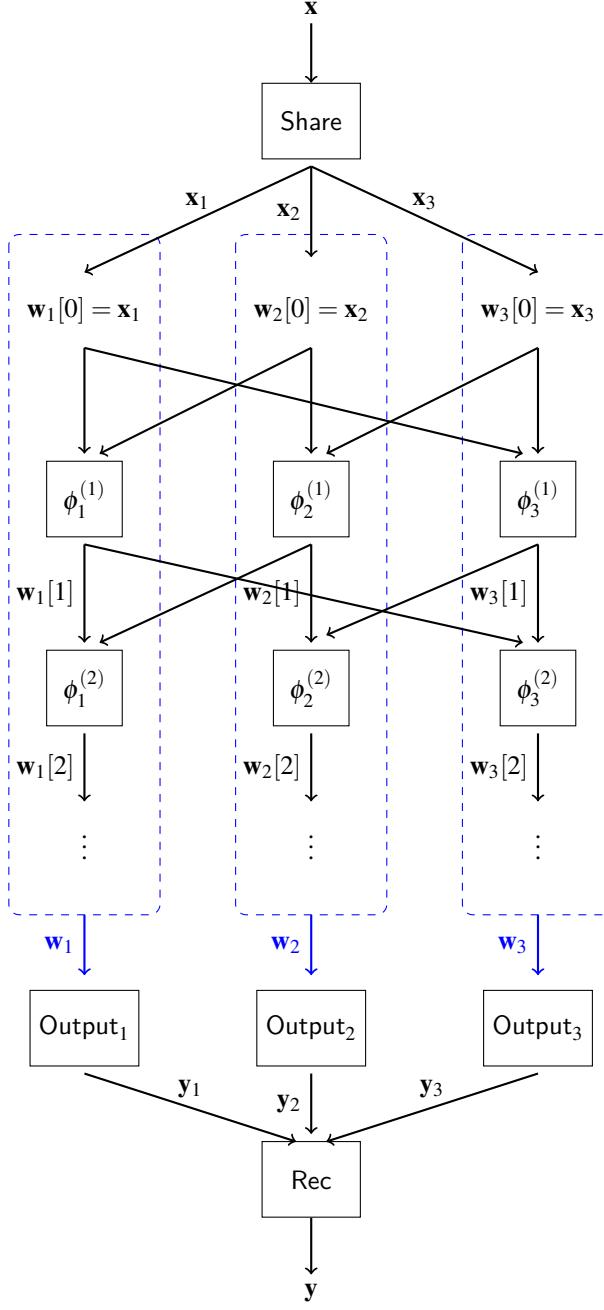


Figure 5: Pictorial representation of a (2,3)-decomposition of the computation $\mathbf{y} = \phi(\mathbf{x})$ showing the three branches.

4.1.1 The Linear Decomposition

We present here an explicit example of a convenient (2,3)-decomposition. Let \mathbb{Z} be an arbitrary finite ring such that $\phi : \mathbb{Z}^k \rightarrow \mathbb{Z}^\ell$ can be expressed by an arithmetic circuit over the ring using addition by constant, multiplication by constant, binary addition and binary multipli-

cation gates⁸. The total number of gates in the circuit is N , the gates are labelled with indices in $[N]$. The *linear* (2,3)-decomposition of ϕ is defined as follows:

- Share $^{\mathbb{Z}}(\mathbf{x}; \mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3)$ samples random $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ such that $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3$;
- The family $\mathcal{F}^{\mathbb{Z}} = \bigcup_{c=1}^N \{\phi_1^{(c)}, \phi_2^{(c)}, \phi_3^{(c)}\}$ is defined in the following way. Assume that the c -th gate has input wires coming from the gate number a and the gate number b (or only gate number a in the case of a unary gate), then the function $\phi_i^{(c)}$ is defined as follows: If the c -th gate is a ($\forall \alpha \in \mathbb{Z}$)
 - unary “add α ” gate, then $\forall i \in [3]$:
$$\mathbf{w}_i[c] = \phi_i^{(c)}(\mathbf{w}_i[a]) = \begin{cases} \mathbf{w}_i[a] + \alpha & \text{if } i = 1 \\ \mathbf{w}_i[a] & \text{else} \end{cases}$$
- unary “mult. α ” gate, then $\forall i \in [3]$:
$$\mathbf{w}_i[c] = \phi_i^{(c)}(\mathbf{w}_i[a]) = \alpha \cdot \mathbf{w}_i[a]$$
- binary addition gate, then $\forall i \in [3]$:
$$\mathbf{w}_i[c] = \phi_i^{(c)}(\mathbf{w}_i[a], \mathbf{w}_i[b]) = (\mathbf{w}_i[a] + \mathbf{w}_i[b])$$
- binary multiplication gate, then $\forall i \in [3]$:
$$\begin{aligned} \mathbf{w}_i[c] &= \phi_i^{(c)}(\mathbf{w}_i[a, b], \mathbf{w}_{i+1}[a, b]) \\ &= \mathbf{w}_i[a] \cdot \mathbf{w}_i[b] + \mathbf{w}_{i+1}[a] \cdot \mathbf{w}_i[b] \\ &\quad + \mathbf{w}_i[a] \cdot \mathbf{w}_{i+1}[b] + R_i(c) - R_{i+1}(c) \end{aligned}$$

where $R_i(c)$ is a uniformly random function sampled using \mathbf{k}_i .

- For all $i \in [3]$, Output $^{\mathbb{Z}}_i(\mathbf{w}_i, \mathbf{k}_i)$ simply selects all the shares of the output wires of the circuit;
- Finally, Rec $^{\mathbb{Z}}(\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3)$ outputs $\mathbf{y} = \mathbf{y}_1 + \mathbf{y}_2 + \mathbf{y}_3$

Proposition 4.1. *The decomposition $\mathcal{D}^{\mathbb{Z}} = \{\text{Share}^{\mathbb{Z}}, \text{Rec}^{\mathbb{Z}}, \text{Output}_1^{\mathbb{Z}}, \text{Output}_2^{\mathbb{Z}}, \text{Output}_3^{\mathbb{Z}}\} \cup \mathcal{F}^{\mathbb{Z}}$ defined above is a (2,3)-decomposition. Moreover, the length of each view in $\mathcal{D}^{\mathbb{Z}}$ is $(k+N+\ell) \log |\mathbb{Z}| + \kappa$ bits.*

Correctness of the decomposition follows from inspection. Privacy can be shown by constructing an appropriate simulator as shown in Appendix A

In the linear decomposition just presented, the parameter N is equal to the total number of gates (unary and binary) in the circuit computing ϕ . It is easy to slightly modify the definition of the functions $\phi_i^{(c)}$ in $\mathcal{D}^{\mathbb{Z}}$ in such

⁸Note that Boolean circuits are a special case of this, with the XOR, AND and NOT gate.

a way that N results equal to the number of multiplication gates only. In particular, note that the evaluation of *addition gates* (both unary and binary) only requires computation on values from the same branch, thus they can be embedded in a generalized multiplication gates which take as input arbitrary subsets of wires A, B , contains constants α, β, γ and computes the value:

$$\mathbf{w}[c] = \left(\sum_{a \in A} \alpha[a] \mathbf{w}[a] \right) \cdot \left(\sum_{b \in B} \beta[b] \mathbf{w}[b] \right) + \gamma$$

4.2 ZKBoo Protocol

Following the idea of [19], we turn a $(2,3)$ -decomposition of a function ϕ into a zero-knowledge protocol for statements of the form “I know \mathbf{x} such that $\phi(\mathbf{x}) = \mathbf{y}$ ”. We indicate with L_ϕ the language $\{\mathbf{y} \mid \exists \mathbf{x} \text{ s.t. } \phi(\mathbf{x}) = \mathbf{y}\}$.

Assume that a $(2,3)$ -decomposition of the function ϕ is known (see Section 4.1). The structure of the resulting protocol (Figure 6) is very similar to the structure of the IKOS protocol. If $\mathbf{y} \in L_\phi$ is the public input of the proof, then the prover P uses his private input \mathbf{x} (with $\phi(\mathbf{x}) = \mathbf{y}$) to run “in his head” the protocol Π_ϕ^* . After the emulation of the protocol, P commits to each of the 3 produced views $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$. Now the verifier challenges the prover to open 2 of the commitments. Finally, the verifier accepts if the opened views are consistent with the decomposition used and with output \mathbf{y} .

Proposition 4.2. *The ZKBoo protocol (Figure 6) is a Σ -protocol for the relation R_ϕ with 3-special soundness.*

Proof. Clearly, the ZKBoo protocol has the right communication pattern and it is complete given that the decomposition \mathcal{D} is correct. Moreover, the protocol satisfies the 3-special soundness property: consider 3 accepting conversations $(\mathbf{a}, i, \mathbf{z}_i)$, $i \in [3]$: first note that thanks to the binding property of the commitment, the view \mathbf{w}_1 contained in \mathbf{z}_1 and the one contained in \mathbf{z}_3 are identical, and the same holds for the other views $\mathbf{w}_2, \mathbf{w}_3$ and random tapes $\mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3$. Then, we can traverse the decomposition of ϕ backwards from the output to the input shares: since the three conversations are accepting, we have that $\text{Rec}(\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3) = \mathbf{y}$, that $\mathbf{y}_i = \text{Output}_i(\mathbf{w}_i) \forall i$, and finally that every entry in all of \mathbf{w}_i was computed correctly. Therefore, since the Share function is surjective, we can compute $\mathbf{x}' = \text{Share}^{-1}(\mathbf{w}_1[0], \mathbf{w}_2[0], \mathbf{w}_3[0])$. Thanks to the correctness of the decomposition we thus have that $\phi(\mathbf{x}') = \mathbf{y}$, which is what we wanted to prove. Note that the protocol does not satisfy 2-special soundness, even if two accepting conversation actually contain all three views: in this case, since one of the branches of

ZKBoo Protocol

The verifier and the prover have input $\mathbf{y} \in L_\phi$. The prover knows \mathbf{x} such that $\mathbf{y} = \phi(\mathbf{x})$. A $(2,3)$ -decomposition of ϕ is given. Let Π_ϕ^* be the protocol related to this decomposition.

Commit: The prover does the following:

1. Sample random tapes $\mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3$;
2. Run $\Pi_\phi^*(\mathbf{x})$ and obtain the views $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$ and the output shares $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3$;
3. Commit to $\mathbf{c}_i = \text{Com}(\mathbf{k}_i, \mathbf{w}_i)$ for all $i \in [3]$;
4. Send $\mathbf{a} = (\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)$.

Prove: The verifier choose an index $e \in [3]$ and sends it to the prover. The prover answers to the verifier’s challenge sending opening $\mathbf{c}_e, \mathbf{c}_{e+1}$ thus revealing $\mathbf{z} = (\mathbf{k}_e, \mathbf{w}_e, \mathbf{k}_{e+1}, \mathbf{w}_{e+1})$.

Verify: The verifier runs the following checks:

1. If $\text{Rec}(\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3) \neq \mathbf{y}$, output reject;
2. If $\exists i \in \{e, e+1\}$ s.t. $\mathbf{y}_i \neq \text{Output}_i(\mathbf{w}_i)$, output reject;
3. If $\exists j$ such that

$$\mathbf{w}_e[j] \neq \phi_e^{(j)}(\mathbf{w}_e, \mathbf{w}_{e+1}, \mathbf{k}_e, \mathbf{k}_{e+1})$$

output reject;
4. Output accept;

Figure 6: ZKBoo protocol for the language L_ϕ in the commitment-hybrid model.

the computation has not been checked, $\exists i$ s.t. \mathbf{w}_i might not be equal to $\phi_i^{(j)}(\mathbf{w}_i, \mathbf{w}_{i+1}, \mathbf{k}_i, \mathbf{k}_{i+1})$.

To prove the special honest-verifier ZK property, we consider the simulator S defined by the following steps. The input are $\mathbf{y} \in L_\phi$ and $e \in [3]$: run the 2-privacy simulator (which is guaranteed to exist thanks to the 2-privacy property of the decomposition \mathcal{D} as in Definition 4.1), which returns $(\{\mathbf{k}_i, \mathbf{w}_i\}_{i \in \{e, e+1\}}, \mathbf{y}_{e+2})$, sets $\mathbf{w}_{e+2} = 0^{|\mathbf{w}|}, \mathbf{k}_{e+2} = 0^{|\mathbf{k}|}$ and then constructs \mathbf{a} by committing to the three views and tapes. \square

Efficiency. Let $\phi : \mathbb{Z}^k \rightarrow \mathbb{Z}^\ell$ be a function that can be expressed by a circuit over the finite ring \mathbb{Z} with N mul-

tiplication gates. If we repeat $\sigma(\log_2 3 - 1)^{-1}$ copies of the ZKBoo protocol instantiated with the linear decomposition described in Section 4.1.1, and where we generate the random tapes pseudo-randomly with security parameter κ , we get a Σ -protocol with soundness $2^{-\sigma}$ and bit-size

$$\sigma(\log_2 3 - 1)^{-1} \cdot 2 \cdot [\log_2(|\mathbb{Z}|)(k + N + \ell) + \kappa]$$

5 Zero-Knowledge for SHA-1/SHA-256

In this section we describe our case study, in which we implemented the protocol described in Section 4 for proving knowledge of preimages of SHA-1 and SHA-256. We start describing the choices we made in our implementation, describe the result of our empirical validation and finally compare with state-of-the-art protocols for the same task. Our implementation is available at <https://github.com/Sobuno/ZKBoo>.

5.1 Circuits For SHA-1/SHA-256

The linear-decomposition protocol described in Section 4 can be used with arithmetic circuits over arbitrary rings. Our first choice is picking a ring in which to express the computation of SHA-1/SHA-256. The two functions are quite similar, and they both use vectors of 32 bits for internal representation of values. Three kind of operations are performed over these bit-vectors: bitwise XORs, bitwise ANDs, and additions modulo 2^{32} . Implementing the two algorithms (after some simple optimization to reduce the number of bitwise ANDs) requires the following number of operations⁹:

	AND	XOR	ADD
SHA-1	40	372	325
SHA-256	192	704	600

Hence, the two natural choices for the ring are \mathbb{Z}_2 (where XOR gates are for free but AND/ADD require 32 multiplication gates) and $\mathbb{Z}_{2^{32}}$ (where ADD is free but bitwise operations require a linear number of multiplication gates). Since the number of XORs dominates in both algorithms, we opted for an implementation over the ring \mathbb{Z}_2 .

5.2 Implementation of Building Blocks

We wrote our software in C, using the OpenSSL¹⁰ library. We instantiated the building blocks in our protocol in the following way:

⁹Note that the AND complexity of our circuits is approximately 1/3 of the “standard MPC circuit” from <https://www.cs.bris.ac.uk/~Research/CryptographySecurity/MPC/>.

¹⁰<https://www.openssl.org>

RNG: We generate the random tapes pseudorandomly using AES in counter mode, where the keys are generated via the OpenSSL secure random number generator. In the linear decomposition of multiplication gates, we use a random function $R : [N] \rightarrow \mathbb{Z}_2$. We implement this function by picking a bit from the stream generated using AES. In particular, we compute

$$R(i) = \text{AES}(K, [i/128])[i \bmod 128]$$

which means that 3 calls to AES are sufficient to evaluate 128 individual AND gates. Note that since N (the number of AND gates) is known in advance, we can precompute all calls to AES at the beginning of the protocol. These two optimizations, together with the native support for AES in modern processors, proved very effective towards decreasing running times.

Commitments: In the first step of the protocol the prover commits to the three views $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$. Those commitments have been implemented using SHA-256 as the commitment function i.e., $\text{Com}(x, r) = \text{SHA-256}(x, r)$. Under the (mild) assumptions that SHA-256 is collision resistant and that $\text{SHA-256}(\cdot, r)$ is a PRF (with key r) the commitments are binding and hiding.

The Fiat-Shamir Oracle. To make the proofs non-interactive, we need a random oracle $H : \{0, 1\}^* \rightarrow \{1, 2, 3\}^r$ where r is the number of repetitions of our basic protocol. We instantiate this using SHA-256 as a random oracle and by performing rejection sampling. In particular, we compute the first output coordinate of $H(x)$ by looking at the first two output bits of $\text{SHA-256}(0, x)$ and mapping $(a, b) \rightarrow 2a + b + 1$. In case that $(a, b) = (1, 1)$ we look at the third-fourth bit instead and repeat. If there are no more bits left in the output of the hash function, we evaluate $\text{SHA-256}(1, x)$ and so on. In our experiments the maximum number of repetition is $r \in \{69, 137\}$, thus we call the hash function once or twice (on expectation).

5.3 Experimental Setup

We report on the results of the implementation of SHA-1 and SHA-256 for 69 and 137 repetitions each. Those correspond to soundness errors 2^{-40} and 2^{-80} . While the security level 2^{-40} is not sufficient for the case of non-interactive zero-knowledge, it offers reasonable security guarantees in the interactive case – note however that in this case our timings are only indicative of the local computation as they do not account for the necessary network communication.

Our experiments were run on a machine with an AMD FX-8350 CPU, running 8 cores at 4.00 GHz. The programs were run under Windows 10 Pro version 1511 (OS Build 10586.14) on a Seagate Barracuda 7200 RPM SATA 3.0 Gb/s hard drive with 16MB cache. Note that

computing and verifying our proofs is an embarrassingly parallel task, thus it was possible to effortlessly take advantage of our multi-core architecture using OpenMP¹¹, an API useful for making a C program multi-threaded. We note that we have only done this for the main loop of the program, which iterates over the individual repetitions of the proofs (which are clearly independent from each other), thus it is likely that there is room for further parallelisation. Timings were done using C native `clock()` function and are measured in milliseconds.

5.4 Experimental Results

Breakdown. In Table 1 we report on the timings we obtained for both SHA-1 and SHA-256, with 69 and 137 rounds, both enabling and disabling parallelisation. In this table we also present a breakdown of the running time. In particular we measure the following phases for the prover:

- *Commit*: This is the time to run the **Commit** procedure (Figure 6) to produce **a**. It is further divided into the following sub-timings: (*Rand. gen.*) Generation of all needed randomness using OpenSSL RNG as well as preprocessing of the PRF; (*Algorithm exec.*) Time taken to run the algorithm Π_ϕ^* . This is the total time for all 69/137 rounds; (*Commitment*) Generating commitments of the views;
- *Gen. challenge*: Using the random oracle to generate the challenge vector as $\mathbf{e} = H(\mathbf{y}, \mathbf{a})$;
- *Prove*: Building the vector **z**;
- *Output to disk*: Writing $(\mathbf{a}, \mathbf{e}, \mathbf{z})$ to disk;¹²

For the verifier:

- *Input from disk*: Reading the proof from file;
- *Gen. challenge*: Regenerate the challenge vector using the random oracle;
- *Verify*: The time to run all the rounds of the **Verify** procedure;

Finally, with *proof size* we indicate the size of the string $\pi = (\mathbf{y}, \mathbf{a}, \mathbf{z})$ on disk in KB.

Parallelisation. Figure 7 and 8 show how the running time of the prover (resp. verifier) changes when we change the number of rounds (from 1 to 137) and the number of threads (from 1 to 8). We include the graphs for SHA-256 only. It is easy to see that the running time

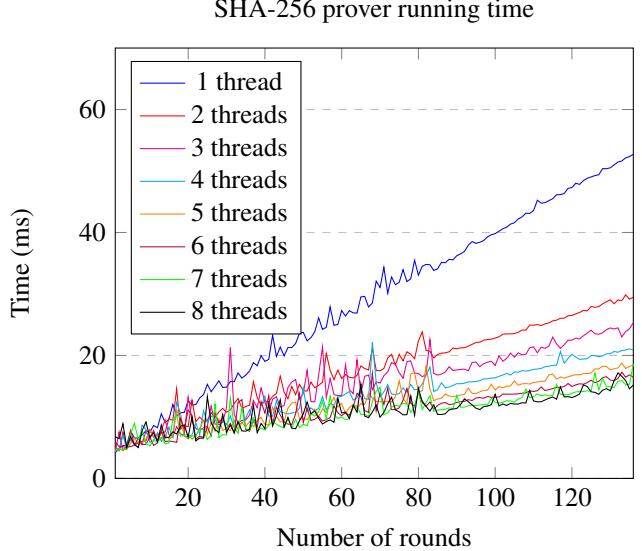


Figure 7: Relation between the total running time and the number of rounds for the SHA-256 prover, average over 100 runs.

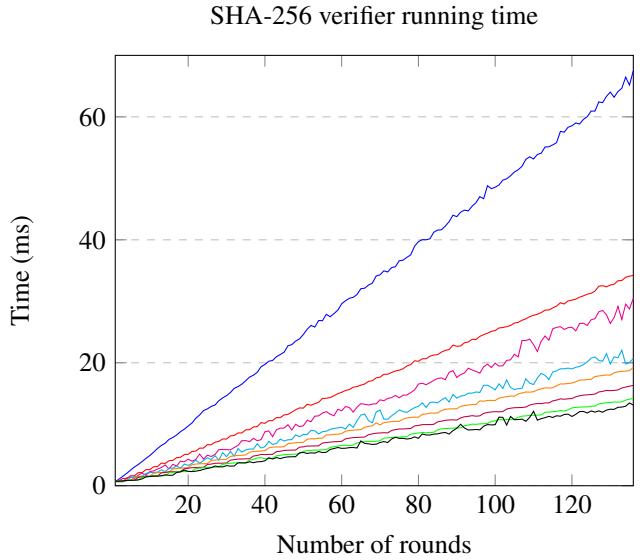


Figure 8: Relation between the total running time and the number of rounds for the SHA-256 verifier, average over 100 runs.

increases linearly with the number of rounds, and that the improvement due to multithreading is significant. The graph indicates that there is some fluctuation in the algorithm's run time for all number of threads when using up to about 85 rounds, which is mostly due to the

¹¹<http://openmp.org>

¹²We observed that the timings of writing to disk are very noisy, and not always monotone in the size of the written file.

	SHA-1				SHA-256			
	69 rounds		137 rounds		69 rounds		137 rounds	
	Serial	Paral.	Serial	Paral.	Serial	Paral.	Serial	Paral.
Prover (ms)	18.98	8.12	31.73	12.73	30.81	12.45	54.63	15.95
Commit	13.45	3.68	26.73	6.59	24.47	5.86	48.25	10.07
- Rand. gen.	1.35	0.60	2.47	0.88	2.28	0.80	4.46	1.13
- Algorithm exc.	10.41	2.69	21.55	5.06	19.60	4.44	38.68	7.87
- Commitment	1.37	0.39	2.71	0.64	2.56	0.62	5.09	1.07
Gen. challenge	0.06	0.05	0.10	0.13	0.05	0.06	0.09	0.09
Prove	0.12	0.18	0.28	0.32	0.32	0.39	0.07	0.53
Output to disk	5.35	4.21	4.62	5.70	5.08	5.39	4.76	4.43
Verifier (ms)	11.68	2.35	22.85	4.39	34.16	6.77	67.74	13.20
Input from disk	0.09	0.11	0.13	0.16	0.15	0.16	0.29	0.25
Gen. challenge	0.06	0.05	0.10	0.10	0.06	0.05	0.10	0.11
Verify	11.53	2.20	22.63	4.12	33.95	6.56	67.35	12.85
Proof size (KB)	223.71		444.18		421.01		835.91	

Table 1: Breakdown of times and proof size for 69/137 rounds of SHA-1/SHA-256, average of 1000 runs

	Preproc. (ms)	Prover (ms)	Verifier (ms)	Proof size (B)
ZKBoo	0	13	5	454840
ZKGC (Estimates)	0	> 19 (OT only)	> 25 (OT only)	186880
Pinocchio	9754	12059	8	288

Table 2: Comparison of approaches for SHA-1

noise introduced by disk operations¹³. We note that the runtime of the verifier benefits more from parallelisation. This is consistent with Amdahl’s law since, as shown in Table 1, the prover spends significantly more time performing tasks which do not benefit from parallelisation (e.g., writing to disk).

5.5 Comparison

Here we compare the performances of ZKBoo with some of the state-of-the-art protocol for the same task. In particular, we compare the performances of proving/verifying knowledge of SHA-1 preimages across ZKBoo, Pinocchio [23] and ZKGC [21].

Pinocchio [23] is an implementation of SNARKs for verifiable outsourcing of computation. While not its main purpose, it can generate zero-knowledge proofs at a negligible extra cost over sound-only proofs. The choice of benchmarking SHA-1 only (and not SHA-256) is due to the fact that the Pinocchio library only contains SHA-1. The runtime reported for Pinocchio are obtained on the same machine as our implementation. The results shows that ZKBoo is faster at both proving and verifying, with an incredible 10^3 factor for the prover. Note

¹³See the full version [14] for graphs showing the running times without disk operations.

here that if the underlying circuit had been larger, the proof size and the verification time of Pinocchio would not change, while its preprocessing and proving time would grow accordingly. We note also that Pinocchio has a large preprocessing time where some prover/verifier key are generated. Those keys are circuit dependent, and for SHA-1 the prover key is 6.5 MB and the verifier key is 1.1 MB. To Pinocchio’s defence, it must be noted that 1) Pinocchio is a general purpose system that can generate proofs for any circuit (provided as an input file) while our implementation contains the SHA circuit hard-coded; 2) according to [23], Pinocchio has not been parallelised; and 3) Pinocchio uses a SHA-1 circuit which is approximately 3 times larger than ours. While it is conceivable that Pinocchio could be made faster using some of the optimizations introduced here, we do not believe that Pinocchio could ever reach proving times similar to ZKBoo, due to the use of heavy public-key technology (exponentiations in a pairing-friendly group) for each gate in the circuit.

ZKGC [21]. For the case of ZKGC, we could not directly compare implementations, since the source code for [21] is not publicly available. In addition, since the publication of [21], several significant improvements have been proposed but have not been implemented yet. Therefore, in Table 2, we give an accurate estimate of

the size of the proofs generated using ZKGC but only a lower-bound for its runtime. The estimates are computed using the following tools: (*GC*) we estimate the proof size using the communication complexity of the most efficient (in terms of communication complexity) garbled circuits, namely *privacy-free garbled circuits* [13, 27] that can be instantiated with as little as one ciphertext (128 bits using AES) per AND gate in the circuit; (*OT*) we plug the size and runtime given by the most efficient OT available [6]. Since the input size of SHA-1 is quite large (512 bits), it might be that using OT extension would prove useful. Therefore, to make the comparison even more favourable towards ZKGC, we only count the runtime of 190 base OTs necessary for active secure OT extensions [1] and we do not account at all for the runtime of the OT extension protocol nor the generation/verification of the GC. The resulting estimates show that even when *counting the base OTs alone*, the runtime of ZKGC is already larger than the runtime of ZK-Boo for the SHA-1 circuit. As for proof size, we note that ZKGC produces shorter proofs. However, the approach of ZKGC cannot be made non-interactive which is a qualitative drawback and it is likely to introduce significant slow-downs due to network latency.

6 Conclusions

In this paper we described ZKBoo, the first attempt to make general purpose zero-knowledge practical using the “MPC-in-the-head” approach of Ishai et al. [19]. We discussed how to generalize their protocol using the idea of $(2, 3)$ -function decompositions, we showed simple linear decompositions for arithmetic circuits over any ring and we leave it as a future work to find compact decompositions for other interesting functions.

Our experimental results show that for practically relevant circuits (such as SHA-1), our protocol is the fastest in terms of proving time, and where the verification time is comparable even with SNARKs technology.

Acknowledgements

This project was supported by: the Danish National Research Foundation and The National Science Foundation of China (grant 61361136003) for the Sino-Danish Center for the Theory of Interactive Computation; the Center for Research in Foundations of Electronic Markets (CFEM); the European Union Seventh Framework Programme ([FP7/2007-2013]) under grant agreement number ICT-609611 (PRACTICE).

References

- [1] ASHAROV, G., LINDELL, Y., SCHNEIDER, T., AND ZOHNER, M. More efficient oblivious transfer extensions with security for malicious adversaries. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I* (2015), pp. 673–701.
- [2] BEN-OR, M., GOLDWASSER, S., AND WIGDERSON, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC* (1988), pp. 1–10.
- [3] BEN-SASSON, E., CHIESA, A., GARMAN, C., GREEN, M., MIERS, I., TROMER, E., AND VIRZA, M. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014* (2014), pp. 459–474.
- [4] BEN-SASSON, E., CHIESA, A., TROMER, E., AND VIRZA, M. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. (2014), pp. 781–796.
- [5] BENDLIN, R., DAMGÅRD, I., ORLANDI, C., AND ZAKARIAS, S. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings* (2011), K. G. Paterson, Ed., vol. 6632 of *Lecture Notes in Computer Science*, Springer, pp. 169–188.
- [6] CHOU, T., AND ORLANDI, C. The simplest protocol for oblivious transfer. In *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings* (2015), pp. 40–58.
- [7] DAMGAARD, I. On σ -protocols (2010). *Lecture on Cryptologic Protocol Theory (Aarhus University, course notes)*.
- [8] DAMGÅRD, I., AND ISHAI, Y. Scalable secure multiparty computation. In *Advances in Cryptology-CRYPTO 2006*. Springer, 2006, pp. 501–520.
- [9] DAMGÅRD, I., ISHAI, Y., AND KRØIGAARD, M. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Proceedings of EuroCrypt* (Springer Verlag 2010), pp. 445–465.
- [10] DAMGÅRD, I., AND ZAKARIAS, S. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC* (2013), pp. 621–641.
- [11] EVEN, S., GOLDRICH, O., AND LEMPEL, A. A randomized protocol for signing contracts. *Commun. ACM* 28, 6 (1985), 637–647.
- [12] FIAT, A., AND SHAMIR, A. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology—CRYPTO’86* (1986), Springer, pp. 186–194.
- [13] FREDERIKSEN, T. K., NIELSEN, J. B., AND ORLANDI, C. Privacy-free garbled circuits with applications to efficient zero-knowledge. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II* (2015), pp. 191–219.
- [14] GIACOMELLI, I., MADSEN, J., AND ORLANDI, C. Zkboo: Faster zero-knowledge for boolean circuits. *Cryptography ePrint Archive*, Report 2016/163, 2016. <http://eprint.iacr.org/>.

- [15] GOLDRICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (1987), ACM, pp. 218–229.
- [16] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof-systems (extended abstract). In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA* (1985), pp. 291–304.
- [17] HAZAY, C., AND LINDELL, Y. *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media, 2010.
- [18] HAZAY, C., AND VENKITASUBRAMANIAM, M. On the power of secure two-party computation. *Cryptology ePrint Archive*, Report 2016/074. To appear in *Crypto 2016*, 2016. <http://eprint.iacr.org/>.
- [19] ISHAI, Y., KUSHILEVITZ, E., OSTROVSKY, R., AND SAHAI, A. Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing* (2007), STOC ’07, ACM, pp. 21–30.
- [20] ISHAI, Y., KUSHILEVITZ, E., OSTROVSKY, R., AND SAHAI, A. Zero-knowledge proofs from secure multiparty computation. *SIAM Journal on Computing* 39, 3 (2009), 1121–1152.
- [21] JAWUREK, M., KERSCHBAUM, F., AND ORLANDI, C. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS’13, Berlin, Germany, November 4-8, 2013* (2013), pp. 955–966.
- [22] NIELSEN, J. B., NORDHOLT, P. S., ORLANDI, C., AND BURRA, S. S. A new approach to practical active-secure two-party computation. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (2012), pp. 681–700.
- [23] PARNO, B., HOWELL, J., GENTRY, C., AND RAYKOVA, M. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), pp. 238–252.
- [24] RABIN, M. O. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive* 2005 (2005), 187.
- [25] RANELLUCCI, S., TAPP, A., AND ZAKARIAS, R. W. Efficient generic zero-knowledge proofs from commitments. *Cryptology ePrint Archive*, Report 2014/934. To appear in ICITS 2016, 2014. <http://eprint.iacr.org/>.
- [26] SCHNORR, C.-P. Efficient identification and signatures for smart cards. In *CRYPTO* (1989), pp. 239–252.
- [27] ZAHUR, S., ROSULEK, M., AND EVANS, D. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II* (2015), pp. 220–250.

A Appendix

Proof of Proposition 4.1

Proof. In order to prove that the decomposition $\mathcal{D}^{\mathbb{Z}}$ is correct is enough to prove that for any $c \in [N]$ the following holds.

(1) if the c -th gate is an “add α ” gate, then:

$$\sum_{i=1}^3 \mathbf{w}_i[c] = \left(\sum_{i=1}^3 \mathbf{w}_i[a] \right) + \alpha$$

(2) if the c -th gate is an “mult. α ” gate, then:

$$\sum_{i=1}^3 \mathbf{w}_i[c] = \left(\sum_{i=1}^3 \mathbf{w}_i[a] \right) \cdot \alpha$$

(3) if the c -th gate is an addition gate, then:

$$\sum_{i=1}^3 \mathbf{w}_i[c] = \left(\sum_{i=1}^3 \mathbf{w}_i[a] \right) + \left(\sum_{i=1}^3 \mathbf{w}_i[b] \right)$$

(4) if the c -th gate is a multiplication gate, then:

$$\sum_{i=1}^3 \mathbf{w}_i[c] = \left(\sum_{i=1}^3 \mathbf{w}_i[a] \right) \cdot \left(\sum_{i=1}^3 \mathbf{w}_i[b] \right)$$

Indeed, using (1), (2), (3) and (4) iteratively for all the gates in the circuit we can prove that $\sum_{i=1}^3 \mathbf{w}_i[N] = \phi(\sum_{i=1}^3 \mathbf{x}_i)$ and from this it follows that

$$\begin{aligned} \Pi_{\phi}^*(\mathbf{x}) = \text{Rec}^{\mathbb{Z}}(\mathbf{y}_1, \dots, \mathbf{y}_n) &= \sum_{i=1}^3 \mathbf{y}_i \\ &= \sum_{i=1}^3 \mathbf{w}_i[N] = \phi \left(\sum_{i=1}^3 \mathbf{x}_i \right) = \phi(\mathbf{x}) \end{aligned}$$

The first three follow trivially by the definition of the function $\phi_{A_i}^{(c)}$ when the c -th gate is a an “add α ”, “mult. α ” and addition gate, respectively. Now assume that the c -th gate is a multiplication gate. Then, using the definition for the function $\phi_{A_i}^{(c)}$ for this case and recalling that the index values are computed modulo 3, we have that

$$\begin{aligned} \sum_{i=1}^3 \mathbf{w}_i[c] &= \sum_{i=1}^3 \left(\mathbf{w}_i[a] \cdot \mathbf{w}_i[b] + \mathbf{w}_{i+1}[a] \cdot \mathbf{w}_i[b] + \mathbf{w}_i[a] \cdot \mathbf{w}_{i+1}[b] \right. \\ &\quad \left. + R_i(c) - R_{i+1}(c) \right) \\ &= \sum_{i=1}^3 \mathbf{w}_i[a] \cdot (\mathbf{w}_i[b] + \mathbf{w}_{i+1}[b]) + \sum_{i=1}^3 \mathbf{w}_i[a] \cdot \mathbf{w}_{i+2}[b] + \\ &\quad + \sum_{i=1}^3 R_i(c) - \sum_{i=1}^3 R_{i+1}(c) \\ &= \left(\sum_{i=1}^3 \mathbf{w}_i[a] \right) \cdot \left(\sum_{i=1}^3 \mathbf{w}_i[b] \right) \end{aligned}$$

We now pass to prove the 2-privacy property. Given $e \in [3]$, we define the simulator S_e on input \mathbf{y} with the following instructions:

1. Sample random tapes $\mathbf{k}'_e, \mathbf{k}'_{e+1}$;
2. Sample uniformly at random the values $\mathbf{w}'_e[0]$ and $\mathbf{w}'_{e+1}[0]$. Then, for all $c \in [N]$: If the c -th gate is an “add α ”, “mult. α ” or addition gate then define $\mathbf{w}'_e[c]$ and $\mathbf{w}'_{e+1}[c]$ using the functions $\phi_e^{(c)}$ and $\phi_{e+1}^{(c)}$, respectively. If the c -th gate is a multiplication gate then sample uniformly at random the value $\mathbf{w}'_{e+1}[c]$ and compute the value $\mathbf{w}'_e[c]$ using $\phi_e^{(c)}$; In this way define the entire views \mathbf{w}'_e and \mathbf{w}'_{e+1} ;
3. Compute $\mathbf{y}'_e = \text{Output}_e(\mathbf{w}'_e)$ and $\mathbf{y}'_{e+1} = \text{Output}_{e+1}(\mathbf{w}'_{e+1})$;
4. Compute $\mathbf{y}'_{e+2} = \mathbf{y} - (\mathbf{y}'_e + \mathbf{y}'_{e+1})$;
5. Output $(\{\mathbf{k}'_i, \mathbf{w}'_i\}_{i \in \{e, e+1\}}, \mathbf{y}'_{e+2})$

It is easy to verify that the output of the simulator S_e has the same distribution of the string $(\{\mathbf{k}_i, \mathbf{w}_i\}_{i \in \{e, e+1\}}, \mathbf{y}_{e+2})$ produced by the protocol Π_ϕ^* . Indeed, all the elements in the output of S_e are computed using the same commands used in Π_ϕ^* , except for the element $\mathbf{w}'_{e+1}[c]$ when the c -th gate is a multiplication gate. In this case $\mathbf{w}'_{e+1}[c]$ is sample uniformly at random, while $\mathbf{w}_{e+1}[c]$ in the protocol is computed using the function $\phi_{e+1}^{(c)}$. In particular, $\mathbf{w}_{e+1}[c]$ is computed by subtracting to determined value the element $R_{i+2}(c)$. Since R_{i+2} is an uniformly random function sampled using an independent tape \mathbf{k}_{e+2} , the distribution of $\mathbf{w}_{e+1}[c]$ in the protocol is the uniform one, that is it has the same distribution of $\mathbf{w}'_{e+1}[c]$ in the output of S_e . Therefore, we can conclude that S_e is a correct simulator for the decomposition $\mathcal{D}^\mathbb{Z}$.

Finally, by inspection we have that $|\mathbf{w}_i| = (k + N + \ell) \log |\mathbb{Z}| + \kappa$ for all $i \in [3]$. \square

The Cut-and-Choose Game and its Application to Cryptographic Protocols

Ruiyu Zhu
Indiana University

Yan Huang
Indiana University

Jonathan Katz
University of Maryland

abhi shelat
Northeastern University

Abstract

The *cut-and-choose* technique plays a fundamental role in cryptographic-protocol design, especially for secure two-party computation in the malicious model. The basic idea is that one party constructs n versions of a message in a protocol (e.g., garbled circuits); the other party randomly *checks* some of them and *uses* the rest of them in the protocol. Most existing uses of cut-and-choose fix *in advance* the number of objects to be checked and in optimizing this parameter they fail to recognize the fact that checking and evaluating may have dramatically different costs.

In this paper, we consider a refined cost model and formalize the cut-and-choose parameter selection problem as a constrained optimization problem. We analyze “cut-and-choose games” and show equilibrium strategies for the parties in these games. We then show how our methodology can be applied to improve the efficiency of three representative categories of secure-computation protocols based on cut-and-choose. We show improvements of up to an-order-of-magnitude in terms of bandwidth, and 12–106% in terms of total time. Source code of our game solvers is available to download at <https://github.com/cut-n-choose>.

1 Introduction

Most efficient implementations for secure two-party computation in the semi-honest setting rely on *garbled circuits*. One party, acting as circuit generator, prepares a garbled circuit for the function of interest and sends it to the other party along with garbled values corresponding to its input. The second party, who will serve as the circuit evaluator, obtains garbled values for its own inputs using oblivious transfer, and then evaluates the garbled circuit to obtain the result.

The primary challenge in handling *malicious* adversaries is to ensure that the garbled circuit sent by the

first party is constructed correctly. The *cut-and-choose paradigm* is a popular and efficient mechanism for doing so. The basic idea is that the circuit generator produces and sends *several* garbled circuits; the circuit evaluator *checks* a random subset of these, and *evaluates* the rest to determine the final result. Since its formal treatment by Lindell and Pinkas [16], numerous works have improved various aspects of the cut-and-choose methodology and used it to design secure protocols [28, 18, 24, 25, 26, 17, 14, 6, 15, 10, 27, 4, 2, 19, 11, 1, 20]. These prior works fall roughly into three categories:

1. **MajorityCut.** Here the circuit evaluator determines its output by taking the majority value among the evaluated garbled circuits. Thus, security holds as long as a majority of the evaluated circuits are correct. This is the classic approach adopted by many papers [16, 28, 18, 17] and implementations [25, 26, 14].
2. **SingleCut.** Here the circuit evaluator is able to obtain the correct output as long as *at least one* of the evaluated circuits are correctly generated. Schemes adopting this approach include [15, 10, 4, 2].
3. **BatchedCut.** This considers a slightly different setting in which the parties repeatedly evaluate some function, and the goal is to obtain good amortized efficiency by batching the cut-and-choose procedure either across multiple instances of secure computation [19, 11], or at the gate level [24, 6].

Although SingleCut is asymptotically better than MajorityCut, some SingleCut protocols [15] require using MajorityCut on a smaller circuit as a sub-routine, and therefore optimizations to MajorityCut can result in efficiency improvements to SingleCut. In addition, MajorityCut works better for applications with long outputs as its cost does not grow with output length.

When setting parameters for cut-and-choose protocols, in order to optimize efficiency for some target level of security, state-of-the-art approaches treat circuit checking roughly as expensive as circuit evaluation, and hence strive to optimize the total number of garbled cir-

Table 1: Bandwidth cost ratios r in various settings.

	AES ^a	Floating pt mult ^b	ORAM R/W ^d	Sort ^c
# AND gates	6800	4300	350 K	6.3×10^9
Ratio r	4533	2866	233 K	4.2×10^9

To be conservative in estimating r , figures assume 128-bit labels. ^aOne-block AES128 with 128-bit wire labels. (Non-free gate counts, 6800, reported in [3, 31], hence $6800 \times 256/(256 + 128) \approx 4533$). ^bA single multiplication of two 64-bit IEEE-754 floating-point numbers [21]. ^cSecurely compute an oblivious access to an ORAM of one million 32-bit numbers [21]. ^dSorting one million 32-bit numbers [21].

cuits involved. Although some researchers [15, 7, 2] observed the asymmetry in the cost of checking and evaluation, they did not explore the cost asymmetry further, and did not investigate the possibility of optimizing cut-and-choose parameters based on this asymmetry.

As evidenced by many recent prototypes [21, 29, 20, 5, 9, 26, 25, 14, 13], network communication has become the most prominent bottleneck of garbled-circuit protocols, especially when exploiting dedicated hardware [3, 8] or parallelism [5, 23, 13] for faster garbling/evaluation. However, the bandwidth costs are markedly different for checking and evaluating circuits: garbled circuits that are evaluated must be transmitted in their entirety, but checking garbled circuits can be done by generating the circuit from a short seed and committing to the circuit using a succinct commitment [7, 14, 2]. Table 1 presents, in the context of a few example applications, the bandwidth costs for sending an entire circuit (i.e. the costs for an evaluated circuit) versus the cost for committing to the circuit (which, for simplicity, requires only one SHA256 hash), and thus a sample ratio r that we use in this paper as a variable.

Based on these observations, we propose a new approach to optimizing parameters in cut-and-choose protocols. Our approach casts the interaction between the circuit generator and circuit evaluator as a game, computes the optimal strategies in this game (which, interestingly, turn out to be mixed strategies), and then sets parameters while explicitly taking into account the relative costs of circuit checking and circuit evaluation. Our optimizations result in cut-and-choose approaches that can be easily integrated into prior protocols, and can reduce the bandwidth in these protocols by an order-of-magnitude in some settings.

1.1 Prior Work

The protocol of Lindell and Pinkas [16] checks exactly half the circuits, an idea followed in many subsequent works [25, 17]. They showed that by generating n circuits and checking a random subset of size $n/2$, a cheating generator succeeds in convincing the evaluator to accept an incorrect output with probability at most $2^{-0.311n}$. Thus, to achieve (statistical) security level 2^{-40} , their protocol requires 128 garbled circuits. Shen and Shelat [26] slightly improved the bound to $2^{-0.32s}$ by opening roughly 60% (instead of one half) of the circuits; this reduces the number of garbled circuits needed to 125 for 2^{-40} security. These protocols belong to the MajorityCut category in our terminology.

The idea of using SingleCut protocols was subsequently introduced [15, 4, 2]. Here, the evaluator chooses whether to check each circuit with independent probability $1/2$; now n circuits suffice to achieve security level 2^{-n} .

Most recently, several works [19, 11, 20] have proposed to amortize the cost of cut-and-choose across multiple evaluations of the same function. Along with the LEGO family of protocols [6, 24] that amortize checks at the gate level (rather than the circuit level), they all fall in the class of BatchedCut protocols. These works show that cut-and-choose can be very efficient in an amortized sense, requiring fewer than 8 circuits per execution to achieve 2^{-40} security when amortizing over 1000 executions. A brief explanation of the BatchedCut idea is given at the beginning of Section 3.3.

1.2 Contributions

We introduce a game-theoretic approach to study cut-and-choose in the context of secure-protocol design. The simplest version of cut-and-choose can be treated as a *zero-sum* game (where the utilities are 0/1 for the loser/winner) between the evaluator and the generator in which the generator wins if it can produce enough incorrect circuits to skew the protocol without being detected. Finding an optimal strategy for the evaluator can be cast as solving a linear-program and results in a randomized strategy for choosing the number of circuits to check. This linear program can be further refined to take into consideration the different cost of checking vs evaluating (i.e., the ratio r). Analyzing the equilibrium of this game leads to a constrained optimization problem that can be used to derive more efficient protocols meeting a targeted security bound (e.g. $\epsilon = 2^{-40}$ as per many published implementations).

Our techniques enable optimization based on the precise relative costs of checking and evaluating, which in turn may depend on the function being computed as well

as characteristics of specific deployment settings, such as software, hardware configuration and network condition, etc. This provides the ability to “tune” protocols to specific applications in a much more fine-grained way than before. We demonstrate that doing so can lead to bandwidth savings of 1.2–10×.

We concretely apply our methodology to three representative types of cut-and-choose-based secure-computation protocols, and show a significant overall improvement in the bandwidth usage. For example, we are able to reduce the network traffic by up to an order-of-magnitude in comparison with the state-of-the-art SingleCut (see Figure 5) and MajorityCut (see Figure 2) protocols, and savings of 20% ~ 80% for state-of-the-art (already highly optimized) BatchedCut protocols (see Figure 8). Our improvements do not require any additional cryptographic assumptions and come with little development overhead.

2 Overview

Notation. Throughout this paper, we implicitly fix the semantic meaning for a few frequently-used variables (unless explicitly noted otherwise) as in Table 2.

Table 2: Frequently-used variables

ϵ	Failure probability of the cut-and-choose game
r	Cost ratio between circuit evaluation and checking
n	Total number of circuit copies ($n = k + e$)
k	Number of circuit copies used for checking
e	Number of circuit copies used for evaluation
b	Number of bad circuit copies generated
T	Total number of circuits used in BatchedCut.
B	Bucket size in BatchedCut.
τ	Evaluator’s detection rate checking a bad gate/circuit

2.1 Problem Abstraction

Let e and k be the numbers of evaluate-circuits and check-circuits, respectively. Let r be the ratio between the costs of evaluating and checking a circuit. In the case when the parameters e, k are set deterministically and public, the cut-and-choose parameter optimization problem can be expressed as the following non-linear programming problem:

$$\arg \min_{e,k} r \cdot e + k$$

subject to

$$\max_b \Pr_a(e, k, b) \leq \epsilon,$$

where ϵ, r are known input constants; $\Pr_a(e, k, b)$ is the probability of a successful attack; and b is the total number of bad circuits generated by the malicious generator.

In the case when at least one of the two parameters (e and k) is randomly picked by the circuit evaluator from some public distributions (but sampled values remain secret to the circuit generator at the time of circuit generation), the optimization problem takes a more general form

$$\arg \min_{S_E} \mathbb{E}[cost(r, S_E)]$$

subject to

$$\mathbb{E}[\Pr_a(S_E, S_G)] \leq \epsilon, \quad \forall S_G$$

where S_E and S_G are the circuit evaluator’s and the circuit generator’s strategies, respectively; $cost$ is the cut-and-choose cost function, and $\mathbb{E}[\cdot]$ denotes the expectation function. Note that the cost function does not need to account for pre-maturely terminated protocol executions (due to detected cheating activity). Our goal is to identify the best S_E for the evaluator. We leave the notion of S_E and S_G abstract for now but will give more concrete representations when analyzing specific protocols in Section 3.

We stress that, in contrast to the common belief used in the state-of-the-art cost analysis of cut-and-choose protocols, the cost of cut-and-choose is usually not best represented by n —the total number of circuits generated, but rather by a cost ratio r between checking and evaluation which depends on many factors such as (1) the kind of cost (e.g., bandwidth or computation); (2) the deployment environment (e.g., network condition, distribution of computation power on the players, buffering, etc.) (3) the specific cryptographic primitives and optimization techniques (e.g., the garbling scheme) used in a protocol. Therefore, the best practice would be always micro-benchmarking the ratio between the per circuit cost of evaluation and checking before running the protocol, and then select the best cut-and-choose strategies accordingly.

2.2 Summary of Our Results

The main thesis of this work is,

Cut-and-choose protocols should be appropriately configured based on the security requirement (ϵ) and the cost ratio (r) benchmarked at run-time. Such practice can bring significant cost savings to many cut-and-choose based cryptographic protocols.

To support our thesis, we have formalized the cut-and-choose-based protocol configuration problem into a constrained optimization problem over a refined cost model.

Our solutions to the constrained optimization problem imply randomized strategies are optimal. We show how to support randomized strategies in the state-of-the-art cut-and-choose-based cryptographic protocols with only small changes. We applied this methodology to analyze three major types of cut-and-choose schemes and the experimental results corroborate our thesis. We have implemented a search tool for each category of schemes to output the optimal parameters. The tool is available at <https://github.com/cut-n-choose>.

3 Case Studies

In this section, we show how our general idea can be applied to three main types of two-party secure computation protocols that are based on the cut-and-choose method to substantially improve their performance. We assume that n is fixed and public, while e will be selected from some distribution and remain hidden to the generator until all circuits are generated and committed.

3.1 MajorityCut Protocols

MajorityCut strategy stems from an intuitive folklore idea: the circuit evaluator randomly selects k (out of a total n circuits) to check for correctness, evaluates the remaining $e = n - k$ circuits, and outputs the *majority* of the e evaluation results. All previous work assumed the use of fixed and public n, e, k parameter values, which grants a malicious generator unnecessary advantages. For example, knowing e , a malicious generator can choose to generate $\lceil e/2 \rceil$ bad circuits to maximize the chance that an honest evaluator outputs a wrong result. Thanks to its simplicity, it is the scheme the most widely adopted by implementations thus far.

In the following, we show how to apply our observations to MajorityCut protocols, which involves delaying the revelation of cut-and-choose parameters and employing a mixed strategy (instead of a pure one) to minimize the total cost of cut-and-choose.

Analysis. We represent the evaluator’s strategy by a vector $\mathbf{x} = (x_0, x_1, \dots, x_n)$ where x_i is the probability that the evaluator evaluates i uniform-randomly chosen circuits and checks the remaining $n - i$. The expected cost of MajorityCut is

$$\sum_{i=0}^n [x_i \cdot (i \cdot r + (n - i))] = n + (r - 1) \sum_{i=0}^n x_i \cdot i$$

If the generator produces b incorrect circuits and the evaluator evaluates i circuits, the probability that the evaluator’s check passes is $\binom{n-b}{n-i} / \binom{n}{n-i}$. After a successful check, the evaluator loses the security game if and only if $2b \geq i$, i.e., there is no majority of correct

evaluation circuits. Hence, when the evaluator uses strategy \mathbf{x} , the expected failure probability of the MajorityCut scheme is

$$\sum_{i \leq 2b} x_i \cdot \binom{n-b}{n-i} / \binom{n}{n-i}$$

Since $i \leq n$ and $\binom{n-b}{n-i} = 0$ for all $i < b$, this sum can be further reduced to $\sum_{i=b}^{\min(n, 2b)} x_i \cdot \binom{n-b}{n-i} / \binom{n}{n-i}$. The security requirement stipulates that for every choice of b by the malicious generator, the resulting cut-and-choose failure probability should be less than ε . In other words, the goal of picking optimal cut-and-choose parameters can be achieved by solving the following linear program:

$$\min_{\mathbf{x}} n + (r - 1) \sum_{i=0}^n x_i \cdot i$$

subject to

$$\begin{aligned} x_i &\geq 0 \\ \sum_{i=0}^n x_i &= 1, \\ \sum_{i=b}^{\min(n, 2b)} x_i \cdot \binom{n-b}{i-b} / \binom{n}{i} &< \varepsilon, \quad \forall b \in \{1, \dots, n\}. \end{aligned}$$

Solving this linear program provides us an equilibrium strategy for every fixed n, ε, r . Using standard LP solvers, such programs can be solved exactly for n that ranges into the thousands (i.e., all practical settings).

With this capability, we can identify, for a given target ε and ratio r , the optimal n (that leads to the least overall cost) by solving the linear programs for all feasible n values. While this leads to the search algorithm described in Figure 1, we note several important observations that speedup the search:

1. We begin our search at $n_0 = \lfloor \varepsilon \rfloor$ and consider $n = n_0, n_0 + 1, \dots$. After solving each LP, we identify a current best cost c^* . Observe that $c^* - (r - 1)$ is an upper-bound of the best n (noted n^*), since any feasible strategy with $n > c^* - (r - 1)$ will cost at least c^* (the evaluator need to evaluate at least one circuit except with at most ε probability). Thus, as our search continues, we update c^* , and terminate the search as soon as all values of n between n_0 and $c^* - (r - 1)$ are examined.
2. When the value of r is beyond moderate (i.e., $r > t_r$ for some constant t_r like 128 with our laptop), searching for the optimal cost becomes time-consuming as it involves solving the above linear programming problem for many relatively large n values (e.g., $n > 500$). In these settings, however, we opt to live with a sub-optimal *pure* strategy, based on the observation that the standard deviation of e is already so small (less

than 0.6 and only keeps decreasing as r grows) that the cost of a sub-optimal pure strategy (i.e. a combination of n and e) approximates the theoretical optimal pretty well (Figure 3b).

3. We note that when $r > t_r$ (step 2), it suffices to search all e less than e_0 (recall (e_0, n_0, c_0) is the starting point of our search, simply derived from the traditional setup with MajorityCut) instead of the infinite range because any strategy with $e = e_0 + 1$ that is more efficient than one with $e = e_0$ has to use at least $r - 1$ fewer check circuits. Since $t_r = 128$, such strategy would have used at least 127 fewer check-circuits, which will contradict with $n_0 = 128$ (assuming $\epsilon = 2^{-40}$).

Results. We have implemented the search algorithm of Figure 1 and run it with a wide range of practically possible r values (see Figure 2). For r values ranging from 5000 to 10^9 , which are typical regarding the cost in network traffic, we can achieve 6 to 16 times savings compared to traditional MajorityCut protocols. Even when r is small, such as $8 \sim 128$ which are representative when considering only the timing cost, our approach brings about 1.45 to 3 times savings. When circuit-level parallelism is exploited like in the work of [14, 13, 5], where r typically ranges from 50 to 500 (see Table 5), we are able to speedup the best existing works by $2.3 \sim 3.9$ times.

Table 3 gives two example optimal strategies for achieving $\epsilon = 2^{-40}$ security when $r = 10$ and $r = 100$, respectively. We observe that the solution mixes fewer pure strategies (which is consistent to the decrease of variance) as r grows. Also note that all pure strategies with even e s are dominated by ones with odd e s. In contrast to current implementations, these strategies suggest that the generator produce a few hundred circuits, but only send roughly 13–21 of them. (Such a scheme is for example quite feasible when using the GPU to produce and commit to garbled circuits.) In comparison, the best protocols that use BatchedCut need to amortize 1000s of protocol executions to achieve security when sending roughly 10 circuits.

In Figure 2, the cross-marked solid curve delineates the optimal cost of mixed strategies (among all strategies with public fixed n), while the dot-marked dashed curve delineates pure-strategy approximation of the optimal mixed strategies (efficiently computed as a result of step 2 of Figure 1 search algorithm). We observe that the pure-strategy approximation actually improves as r get bigger. But when r is relatively small ($100 \geq r \geq 1$), our optimization-based approach can indeed bring about $1\% \sim 11\%$ extra improvement (Figure 2). Last, the curves for two different ϵ values have similar shape but the improvement as a result of our approach is significantly larger for smaller ϵ values.

We also observe from Figure 2 that the performance

Table 3: Example optimal strategies for MajorityCut protocols. ($\epsilon = 2^{-40}$, only non-zero x_i s are listed.)

$r = 10$			$r = 100$		
n	i	x_i as %	n	i	x_i as %
361	7	$1 \cdot 10^{-4}$	514	3	$4 \cdot 10^{-6}$
	9	$9 \cdot 10^{-4}$		5	$2.04 \cdot 10^{-4}$
	11	$7 \cdot 10^{-3}$		7	$7.44 \cdot 10^{-3}$
	13	$4.54 \cdot 10^{-2}$		9	0.21
	15	0.25		11	4.86
	17	1.23		13	94.73
	19	5.36		15	0.19
	21	20.9		23	72.2
Saves 13.5% b/w			Saves 65.3% b/w		

boost seems to be upper-bounded by some value related to ϵ , no matter how big r becomes. This makes some intuitive sense because the cost of our optimized protocols will be upper-bounded by a linear function of r (as the solution comes out of solving the linear programming problem where r constitutes the coefficients of the unknowns). We leave the formal proof of this intuition as an interesting future work.

To examine the characteristic of our solution more closely for $1 \leq r \leq 128$, we have plotted the comparison of overall cost of the optimal strategy with respect to best prior works (Figure 3a), the standard deviation of the overall cost (Figure 3b, recall the optimal strategy is a randomized strategy), and the best n used in every optimal strategy (Figure 3c). Note that the standard deviation of the overall cost is also exactly the standard deviation of e because the randomness in cost all comes from the randomness in selecting e . The fact that the standard deviation quickly drops to less than 0.3 (when $r \geq 100$) and strictly decreases justifies the accuracy of pure strategy approximation for large r .

Changes to Existing Protocols. Our approach to MajorityCut applies to many published cut-and-choose based two-party computation protocols; in particular, it applies directly to those protocols in which the generator first commits to n garbled circuits, and later, after a coin-tossing protocol between generator and evaluator, opens each check circuit by sending either (a) both the 0 and 1 labels for all of its input wires, or (b) the random coins used to construct the circuit. Goyal, Smith and Mohassel [7] were the first to use this technique and the second

Input: ϵ, r .
Output: c^*, n^*, \mathbf{x}^* .

1. If $r \leq t_r$,
 - (a) Initialize $n_0 := \lfloor \log \epsilon \rfloor$ and $c^* := +\infty$.
 - (b) For $n := n_0$ to $c^* - (r-1)(1-\epsilon)$,
 - i. Solve the MajorityCut linear programming problem for (n, ϵ, r) to obtain (c, \mathbf{x}) where c is the minimal cost of $\text{LP}(n, \epsilon, r)$ and \mathbf{x} represents the corresponding strategy to achieve c .
 - ii. If (c, \mathbf{x}) is a feasible solution and $c^* > c$, then $(c^*, \mathbf{x}^*, n^*) := (c, \mathbf{x}, n)$.
 - (c) Output (c, n^*, \mathbf{x}^*) .
2. If $r > t_r$,
 - (a) Initialize $n_0 := 3 \lfloor \log \epsilon \rfloor, e_0 = n_0/2, c_0 := 3 \lfloor \log \epsilon \rfloor + (r-1)e_0$ and $c^* = c_0, n^* = n_0$.
 - (b) For $i := 1$ to $+\infty$ until $e_{i-1} = 1$,
 - i. Set $e_i = e_{i-1} - 1$ and compute the smallest (c_i, n_i) that satisfies the security constraints.
 - ii. If $c^* > c_i$, then $(c^*, n^*, e^*) := (c_i, n_i, e_i)$.
 - (c) Output $(c, n^*, \{x_0, \dots, x_n\})$ where $x_i = 1$ if $i = e^*$, and $x_i = 0$ otherwise.

Figure 1: Search the most efficient strategy (n, \mathbf{x}) for MajorityCut protocols. $\log(\cdot)$ is base-2. c is the minimal cost, n is the fixed total number of circuits and $\mathbf{x} = (x_0, \dots, x_n)$ stands for the evaluator’s best strategy to sample e . While the value of t_r depends on hardware and users’ tolerance of performance. $t_r = 128$ works well on a MacBook Air for $\epsilon = 2^{-40}$.

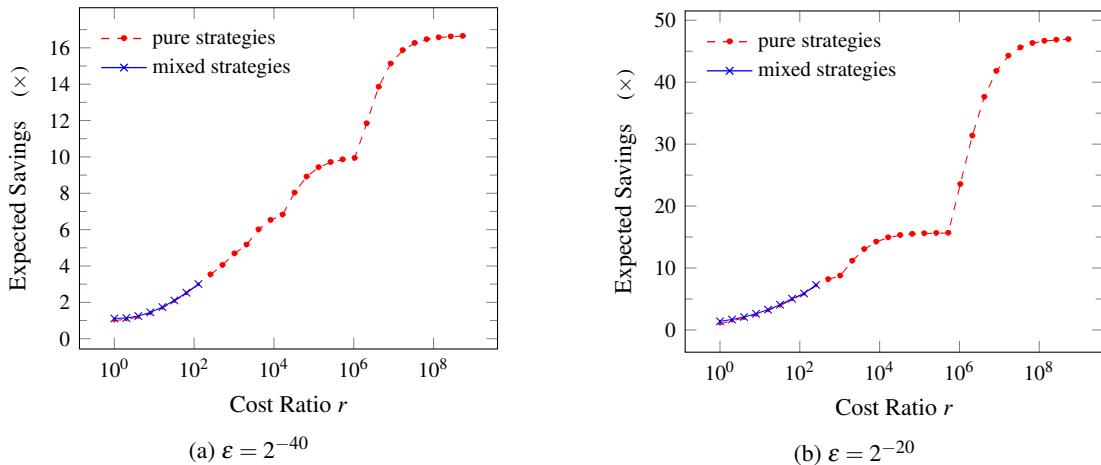


Figure 2: Our savings for MajorityCut protocols

(“ $I + 2C$ ”) protocol from Kreuter, Shelat, Shen [14] also operates in this way. In these cases, no modifications to the security analysis are needed. For every specific ϵ and r , our solver outputs a particular n and a distribution \mathbf{x} for picking e . Roughly speaking, the only changes needed in the protocol are straightforward: the evaluator announces this n beforehand and the result of the coin-tossing protocol ρ is used to sample e according to \mathbf{x} using standard methods (instead of the $1/2$ or $3/5$ fractions as before). The simulation of a malicious evaluator proceeds as in the original security proof with the exception that the simulator first samples e according to \mathbf{x} using random tape ρ and then (as before), uses a simulated coin-tossing to ensure the outcome of the toss induces ρ . (The coin-tossing method is a simple and effective method to prove security; other proofs may also exist.)

Similarly, the first protocol of Lindell and Pinkas [16]

can be modified to adopt this idea: step (3) should send commitments to garbled circuits, modify step (4) to use the random tape from coin-tossing to sample e , modify step (8) so that the garbler sends the entire garbled circuits for the evaluation specimens as well as openings to the commitments so that the evaluator can check consistency.

The idea seems applicable to many protocols which have the property that the set of checked circuits becomes publicly verifiable. For example, Mohassel and Riva [22] use a different idea in their protocol to allow the same output labels to be used across all n copies of the garbled circuit. In their original protocol, the evaluator and generator then use coin-tossing to select the open circuits, but then proceed to evaluate the remaining circuits first, perform some checks, and then the evaluator commits to the output labels. Finally, the generator opens the check

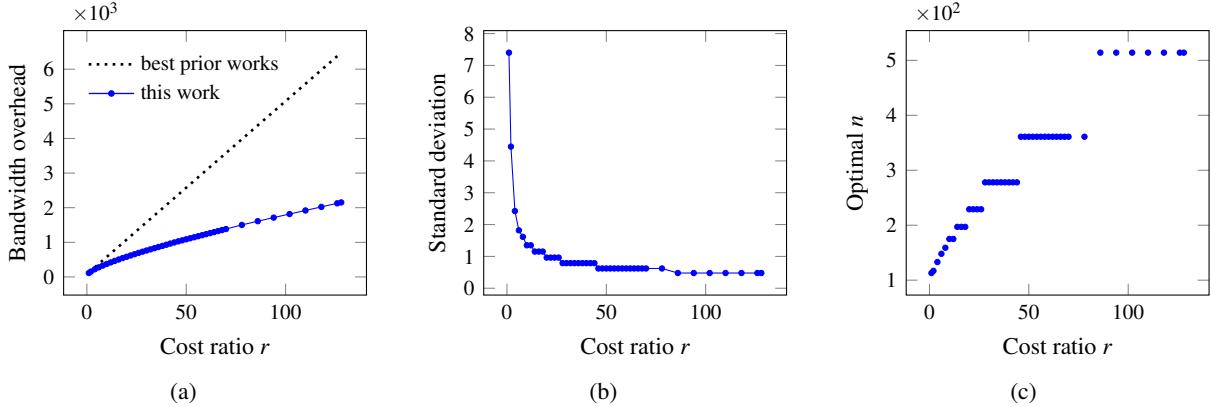


Figure 3: Characteristics of optimal mixed-strategy solutions for MajorityCut protocols ($\varepsilon = 2^{-40}$). The bandwidth overhead is measured in *units*. A unit cost is that of evaluating a evaluation-circuit. The standard deviation chart applies to both the overall cost and e .)

circuits for the evaluator to check, and if all succeed, the evaluator opens a commitment to the output. Although a different order, the modifications noted above seem to apply without the need to modify the security proof.

The protocols of Lindell and Pinkas [17] and shelat and Shen [27], however, seem to require more subtle modifications and new security arguments to use our technique. In both cases, the protocols use a special oblivious transfer (instead of coin-tossing) to allow the evaluator to independently choose the set of check circuits. In the case of [17], the fact that the size of the set of checked circuits is *fixed*, and therefore verifiable by the garbler, is needed in the security proof. This restriction can be lifted with a variant of cut-and-choose oblivious transfer proposed and used in Lindell’s Single-Cut protocol [15]. For a different reason, a new security argument will also be needed for shelat and Shen [27].

3.2 SingleCut Protocols

With SingleCut protocols, extra cryptographic mechanisms (e.g., a second-stage fully secure computation as in [15] or an additive homomorphic commitment as in [2]) are employed in order to *weaken* the requirement for the soundness property. In particular, in such protocols, it suffices to ensure that *at least one evaluation circuit selected by the evaluator is not corrupted*. If one evaluation circuit is properly formed, then the evaluator will either receive the same output from all of the evaluated circuits (in which case it can accept the output since one circuit is good), or it receives two different outputs. In the latter case, the evaluator uses the two different authenticated output labels to recover the garbler’s input, and then evaluate the function itself.

The state-of-the-art SingleCut protocols implicitly assume $r = 1$, in which case an honest evaluator’s best

strategy is to *evaluate each garbled circuit with probability 1/2*, as there is only a single way for the malicious generator to win the cut-and-choose game. In reality, however, r is not necessarily equal to 1. In order to achieve ε statistical security, this strategy will lead to an expected $\lfloor \log \varepsilon \rfloor \cdot (r+1)/2$ units of cost.

Analysis. As before, let i be the number of evaluation circuits and x_i is the probability that the evaluator chooses to evaluate i circuits. An evaluator’s strategy is denoted by $\mathbf{x} = \{x_0, \dots, x_n\}$. Then the cost of the cut-and-choose scheme is $n + (r-1) \sum_{i=0}^n x_i \cdot i$.

Fix b , the number of incorrect circuits chosen by the generator. The first observation is that when the evaluator picks $e \neq b$, then the generator certainly loses the game. When $e = b$, recall that there are $\binom{n}{b}$ different ways to select b evaluation circuits (out of n circuits in total). Assuming the evaluator uniform-randomly picks one of the $\binom{n}{b}$ ways, then the generator looses the cut-and-choose game with probability $1/\binom{n}{b}$ because it happens only if the generator guesses all n of the evaluator’s check-or-evaluate decisions correctly. Since the event that the evaluator picks $e = b$ is independent of the event that the generator guessed all decisions correctly, the overall failure probability is $x_b / \binom{n}{b}$. As a result, the security requirement can be dramatically simplified in comparison to MajorityCut. In particular, we need that every pure strategy for the generator, i.e., every choice of b , wins with probability at most ε : $x_b / \binom{n}{b} < \varepsilon$.

Therefore, fixing n , r and ε , the original cut-and-choose game configuration problem can be translated into the following linear programming problem:

$$\min_{\mathbf{x}} n + (r-1) \sum_{i=0}^n x_i \cdot i$$

Input: ε, r .
Output: c^*, n^*, \mathbf{x}^*

1. Initialize $n_0 := \lfloor \log \varepsilon \rfloor$ and $c^* := n_0 \cdot (r + 1)/2$.
2. For $n := n_0$ to $c^* - (r - 1)(1 - \varepsilon)$,
 - (a) Solve the SingleCut linear programming problem for (n, ε, r) to obtain (c, \mathbf{x}) where c is the minimal cost of LP(n, ε, r) and \mathbf{x} represents the corresponding strategy to achieve c .
 - (b) If $(c, \mathbf{x}) \neq \perp$ and $c^* > c$, then $(c^*, \mathbf{x}^*, n^*) := (c, \mathbf{x}, n)$.
3. Output (c, n^*, \mathbf{x}^*) .

Figure 4: Search the optimal strategy (n, \mathbf{x}) for SingleCut protocols. $\log(\cdot)$ is base-2. c is the minimal cost, n is the fixed total number of circuits and $\mathbf{x} = (x_0, \dots, x_n)$ stands for the evaluator’s best strategy to sample e .

subject to

$$\begin{aligned} x_i &\geq 0, \quad \forall i \in \{0, \dots, n\} \\ \sum_{i=0}^n x_i &= 1, \\ x_b / \binom{n}{b} &< \varepsilon, \quad \forall b \in \{0, \dots, n\}. \end{aligned}$$

Next, we show that the linear programming problem above can actually be solved highly efficiently thanks to its special form. The key observation is that this linear programming problem is in essence a special *continuous knapsack program* (where the weight $w_i = i$). In order to minimize $\sum_{i=0}^n x_i \cdot i$, we aim to maximize x_i (which is upper-bounded by $\varepsilon \cdot \binom{n}{i}$ and collectively constrained by $\sum_{i=0}^n x_i = 1$) for all small i ’s. This leads to the following simple greedy algorithm that solves the problem in linear time (of n).

1. For $i = 0$ to n ,
 - (a) Set $x_i := \varepsilon \cdot \binom{n}{i}$.
 - (b) If $\sum_{j=0}^i x_j \geq 1$ then set $x_i := 1 - \sum_{j=0}^{i-1} x_j$, $x_j := 0$ for all $j > i$, and return $\{x_i | 0 \leq i \leq n\}$.
2. If $\sum_{j=0}^i x_j < 1$, return \perp (i.e., the problem has no feasible solution); otherwise, return $\{x_i | 0 \leq i \leq n\}$.

As with the MajorityCut setting, we scan all possible values of n to identify the best n leading to the smallest overall cost (Figure 4). Fortunately, thanks to this highly efficient special solver, we are always able to identify the best n within seconds for r as large as 10^{10} .

Results. Using the search algorithm described above, we are able to compute the fixed- n , variating e optimal randomized strategies for every ε and r . We summarize the performance gains in Figure 5a. The savings due to our approach rise steadily for $r < 10^4$ and can get to about 10X for reasonably large r (e.g., $r = 7 \times 10^7$, which roughly corresponds to the bandwidth-based cost-ratio for privately computing the edit distance between

two 1000-character strings). Generally, it appears that the improvement-curves (Figure 5) for different ε share some similarity in their shape but smaller ε results in bigger improvements.

Table 4 shows two example optimal strategies of SingleCut protocols for $r = 10$ and $r = 100$, respectively. We observe that the optimal strategy exhibit some pattern: the number of evaluation circuits with positive support falls within $[0, g]$ where $g < n$ and g shrinks as r grows. An interesting note is that $e = 0$ (i.e., checking all n circuits) has positive support, albeit with probability less than 2^{-40} (note the “%” sign), hence preserving security. Instead of artificially preventing $e = 0$ as Lindell did [15], our solution indicates that a rational evaluator should set $e = 0$ with some negligible probability to maximize its chance to win (while keeping the expected cost low).

Table 4: Example optimal strategies for SingleCut protocols. ($\varepsilon = 2^{-40}$, only non-zero x_i ’s are listed)

$r = 10$			$r = 100$		
n	i	x_i as %	n	i	x_i as %
65	0	$9 \cdot 10^{-11}$	180	0	$9 \cdot 10^{-11}$
	1	$5.91 \cdot 10^{-9}$		1	$1.64 \cdot 10^{-8}$
	2	$1.89 \cdot 10^{-7}$		2	$1.47 \cdot 10^{-6}$
	3	$3.97 \cdot 10^{-6}$		3	$8.69 \cdot 10^{-5}$
	4	$6.16 \cdot 10^{-5}$		4	$3.85 \cdot 10^{-3}$
	5	$7.51 \cdot 10^{-4}$		5	0.14
	6	$7.51 \cdot 10^{-3}$		6	3.95
	7	$6.33 \cdot 10^{-2}$		7	95.91
	8	0.46			
	9	2.91			
	10	16.28			
	11	80.28			
			Saves 26.4% b/w		
			Saves 57.0% b/w		

Figure 6 presents a closer look at various characteristics of the optimal strategies, including expected costs (Figure 6a), the standard deviation of the costs (which also applies to e , Figure 6b), and the best ns associated with those optimal strategies (Figure 6c). We note in Figure 6b that the standard deviation for SingleCut optimal strategies are generally smaller (about half) than that for MajorityCut strategies (Figure 3b). In addition, the ns for the optimal strategies also exhibit a staircase effect like in MajorityCut. This is because for any fixed ε , it does not make sense to trade in a larger n for a smaller e ,

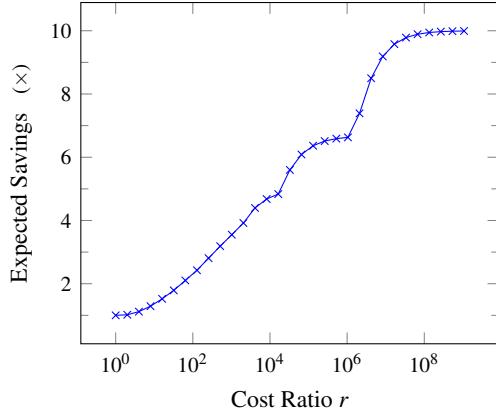
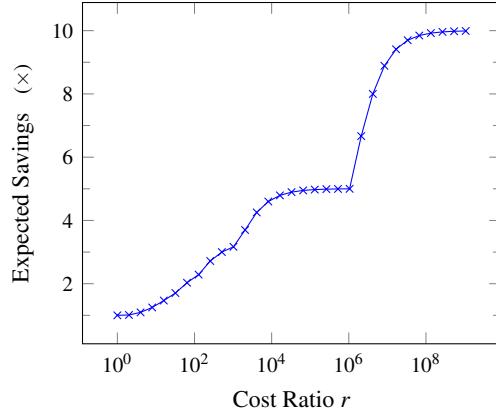
(a) $\epsilon = 2^{-40}$ (b) $\epsilon = 2^{-20}$

Figure 5: Bandwidth savings for SingleCut protocols

unless r exceeds certain discrete threshold values.

Changes to Existing Protocols. The changes needed in protocol 3.2 of Lindell (CRYPTO, 2013) to support our technique are standard:

- Add a step 0 to [15, Protocol 2], where an n is fixed in advance based on ϵ and r .
- Change step 2(b) of [15, Protocol 2] to: P_2 picks the check-set J at random so that $|J| = k$, where k is randomly sampled from the distribution computable (from n, r, ϵ) by the 2-step algorithm given above.

Afshar et al. [2] present a conceptually simple and elegant non-interactive secure computation protocol; it uses a cut-and-choose technique and achieves security 2^{-40} by sending 40 garbled circuits. Surprisingly, this can be done in just one round. Like Lindell, they create a trapdoor which allows an evaluator to recover the garbler’s inputs with high probability if inconsistent but valid output wire-labels are obtained. However, since their protocol is only 1 round, the cut-and-choose is implemented through oblivious transfer; specifically, the evaluator recovers a seed for all of the check circuits through OT, and the garbler sends all circuits in its one message. In order to apply our technique, we need to add extra rounds (in order to save substantial communication costs). Instead of sending the full circuits in the first message, we change the protocol to send succinct commitments of the circuits (thereby committing the sender) which keeps the first message short. In the next message, the evaluator asks the garbler to send the evaluation circuits only; and the evaluator uses its previous messages to continue running the original protocol. We believe these modifications are consistent with the security proof implicitly given in [2]. As a result, we can run this protocol with significantly less communication when $r > 1$.

3.3 BatchedCut Protocols

The basic idea of BatchedCut is to amortize the cost of cut-and-choose across either many protocol executions (of the same circuit) [11, 19] or many basic gates [12, 6, 24] of a big circuit. Without loss of generality, we focus on the setting of batched execution of a single functionality. Roughly speaking, the evaluator randomly selects and checks k out of T circuits in total and randomly groups the remaining circuits in buckets of size B . The state-of-the-art can ensure correctness as long as at least one good circuit is included in every bucket. This can effectively reduce the number of circuit copies to less than 8 (c.f. the optimal 40 without amortization [2, 15]) per execution to ensure 2^{-40} security. However, optimality of this result holds only if $r = 1$. In this section, we present our approach to optimize BatchedCut protocols for general r values.

We note one technical complication in this setting: in the checking stage, a bad circuit (or gate) might only be detected by the evaluator with probability τ . Although $\tau = 1$ for most protocols, it can be less than 1 for some other protocols, e.g., $\tau = 1/2$ for [12] and $\tau = 1/4$ for MiniLEGO [6]. Our analysis below is generalized to account for any $0 \leq \tau \leq 1$. State-of-the-art BatchedCut protocols [19, 11, 12] only require one object in each bucket to evaluate being correct, hence the focus of our analysis.

Analysis. Let N be the number of times a particular functionality will be executed, T be the total number of circuits generated to realize the N executions, and let B denote the bucket size.

With any positive r , we want to identify parameters (T, B) such that $cost(T, B)$ is minimized over all (T, B) configurations that satisfy the security constraint. That is, Pr_{fail} , the overall failure probability of cut-and-choose, should be no more than ϵ . Therefore, the problem reduces to the following constrained optimization

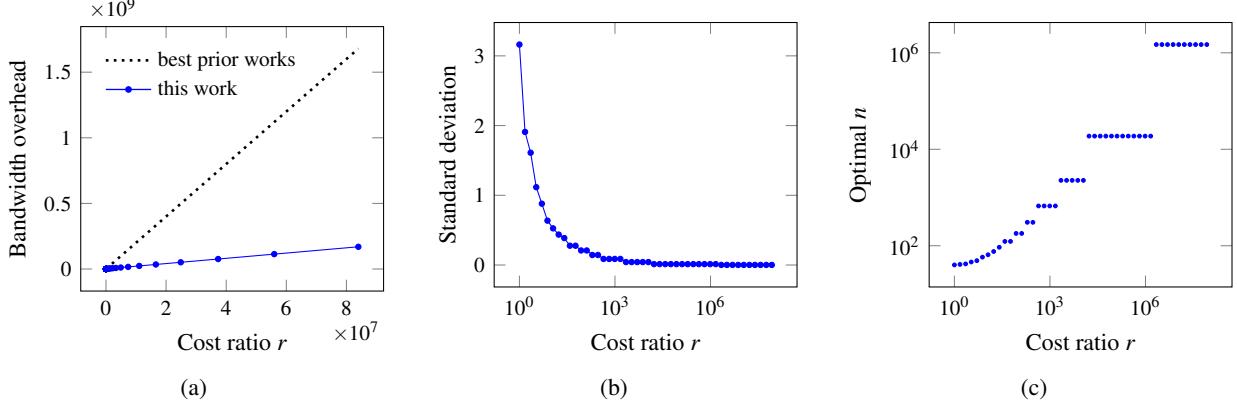


Figure 6: Characteristics of optimal mixed-strategy solutions for SingleCut protocols ($\varepsilon = 2^{-40}$). The bandwidth overhead is measured in *units*. A unit cost is that of evaluating a evaluation-circuit. The standard deviation chart applies to both the overall cost and e .)

problem:

$$\min T + (r - 1)BN$$

subject to

$$\Pr_{fail}(N, T, B, \tau, b) < \varepsilon, \quad \forall b \in \{0, \dots, T\}$$

where b is the number of bad circuits a malicious generator chooses to inject.

Note that \Pr_{fail} describes the failure across all N executions as follows: In the first move of the game, the evaluator picks $T - BN$ circuits to open and verifies all are correct. In the second move, the evaluator randomly partitions the BN unopened circuits into buckets of B circuits. A failure occurs if (i) the adversary is able to corrupt b circuits such that the first check passes, and (ii) there is some bucket containing only corrupted circuits. We let $\Pr_c(N, B, T, \tau, b, i)$ denote the probability of (i) and $\Pr_e(N, B, b)$ denote that of (ii).

First, as before, when i circuits are opened in the first phase, the garbler succeeds with probability

$$\Pr_c(N, B, T, \tau, b, i) = (1 - \tau)^i \binom{b}{i} \binom{T - b}{T - BN - i} / \binom{T}{T - BN}.$$

Here the extra $(1 - \tau)$ term reflects the case when checking a circuit can succeed with some chance even if it is corrupt. There are T total circuits, and $T - BN - i$ of them can be checked. The next term, \Pr_e reflects the probability that conditioned on the first phase passing, the evaluator randomly assigns the remaining circuits to buckets, and one bucket of size B contains all corrupted circuits.

$$\Pr_e(N, B, b) = 0, \quad \forall 0 \leq b \leq B \tag{1}$$

$$\Pr_e(N, B, b) = \binom{b}{B} / \binom{BN}{B} + \sum_{i=0}^{b-1} \Pr_e(N-1, B, b-i) \cdot \frac{\binom{b}{B-i} \binom{BN-b}{i}}{\binom{BN}{B}} \tag{2}$$

The first equation holds because a garbler who corrupts fewer than B circuits never succeeds. Finally, since phase

1 and phase 2 are independent, we conclude that

$$\Pr_{fail}(N, B, T, \tau, b) = \sum_{i=0}^b \Pr_c(N, B, T, \tau, b, i) \Pr_e(N, B, b-i)$$

The summation over i occurs because every check only succeeds with probability τ , and thus even after i checks on corrupted circuits, $b - i$ corrupted circuits may remain in the second phase.

Having explained the constraint, Figure 7 describes our search algorithm to solve the BatchedCut parameter optimization problem. The basic idea is simple—for every $B = 2, 3, \dots$, find the least T such that the security constraint is satisfied for every $b \in \{0, \dots, T\}$. Our main contribution here is to make the search efficient enough for realistic r, N , and ε , which is achieved based on a new efficient and accurate way to calculate \Pr_{fail} and the following observations to ensure efficiency and completeness of the search:

- For every B , the cost $cost(T, B)$ strictly increases with T while the failure rate \Pr_{fail} strictly decreases with T . So the best T for a given B can be identified efficiently using a combination of *exponential backoff* and *binary search*.
- The constraint that $\Pr_{fail} < \varepsilon$ regardless of the attacker’s strategy can be verified by computing \Pr_{fail} for every $b \in \{1, \dots, T\}$ (where b is the number of corrupted circuits generated by the attacker), which, if naively implemented, would require computing \Pr_{fail} $T \cdot T$ times for every B . We can leverage the idea of generating functions to reduce it to $T + T$ inexpensive operations (we will detail this in a bit).
- Assuming $c = (T - BN)/T > c_0$ (where c_0 is a small positive constant determined solely by the evaluator), it does not make sense for a malicious generator to insert more than $b^u = -(s+1)/\log(c_0/2 + 2/(1-c_0) - i_0/N)$ bad circuits. We shall prove this obser-

- vation as Claim 2. This observation further reduces $T + T$ down to $b^u + b^u$ inexpensive operations.
4. Assuming $(T - BN)/T > c_0$, a smaller feasible T identified during the search stipulates an upper-bound on the Bs that needs to be examined.

Compute Pr_{fail} Efficiently. For every N, B, T, τ, b , the probability of a malicious generator's successful attack can be described by equations described above. However, for most N, T values (e.g., $N > 2^{15}$), it is infeasible to compute Pr_c (which involves calculating large binomial coefficients) and Pr_e (which involves exponential number of slow recursions) accurately based on (3.3) and (2).

Hence, we propose an efficient way to compute Pr_e and Pr_c with provable accuracy.

1. **Compute $\text{Pr}_e(N, B, b)$.** The idea is to use *generating functions* to efficiently calculate Pr_e as the ratio between the number of ways to group garbled circuits into buckets that will result a failure (i.e., at least one bucket is filled with all B bad circuits) and the total number of ways to group the garbled circuits. First, we use function $g(x, y) = (1+x)^B + (y-1)x^B$ to model the circuit assignment process for a single bucket, where 'x' denotes a "bad" gate and '1' denotes a "good" gate, thus the coefficient of x^i in $g(x, y)$ equals to the number of ways to assign i bad gates to a bucket. Note that the symbol 'y' we intentionally introduce as the coefficient of x^B term of $g(x, y)$ denotes the event that "all B gates in a bucket are bad". Next, we use the generating function $G(x, y) = g(x, y)^N$ to model the circuit assignment process over all of the N buckets: the coefficient of x^i (which is a polynomial in y , hence written as $f_i(y)$) in $G(x, y)$ denotes the number of assignments involving i bad gates. Let $f_i(y) = \sum_{j=0}^{\infty} c_j y^j$ (where c_j are constants efficiently computable from $G(x, y)$), then $f_b(1) = \sum_{j=0}^{\infty} c_j$ is the total number of assignments with b bad gates used in the evaluation stage; and $f_b(1) - f_b(0) = \sum_{j=1}^{\infty} c_j$ is the number of assignments (among all with b bad circuits) that result in at least one broken bucket. Hence, we compute $\text{Pr}_e(N, B, b) = (f_b(1) - f_b(0))/f_b(1)$. Further, we can dramatically reduce the cost of computing the coefficients of $G(x, y)$ by not distinguishing any terms y^{j_1} and y^{j_2} for any $j_1, j_2 \geq 1$. That is, multiplying $(u+vy)$ and $(w+ty)$ yields $uw + (ut + vw + vt)y$, hence, however big N and B are, $f_i(y)$'s are linear formulas in y .
2. **Compute $\text{Pr}_c(N, B, T, \tau, b, i)$.** Recall that typically T, N are large while b, i are far smaller than N . So the dominating cost in computing Pr_c is to calculate $\left(\frac{T-b}{T-BN-i}\right) / \left(\frac{T}{T-BN}\right)$. To this end, we approximate $\text{Pr}_c(N, B, T, \tau, b, i)$ using $\left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i}\right)^{b-i}$, whose high accuracy is formally proved in Claim 1.

To illustrate the precision of the above calculation, e.g., when $s = 40$, if $N = 50,000$, the overall error in our calculation of $\log \text{Pr}_{fail}(N, B, T, \tau, b)$ is less than 1. Note the error only decreases as N grows (following Claim 1 and Claim 2).

Claim 1 *Let T, B, N, b, i be defined as above. Then*

$$\lim_{N \rightarrow \infty} \frac{\left(\frac{T-b}{T-BN-i}\right)}{\left(\frac{T}{T-BN}\right)} = \left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i}\right)^{b-i}.$$

Claim 2 *Let $\text{Pr}_{fail}(N, B, T, \tau, b)$ be the probability that the cut-and-choose game fails in a BatchedCut scheme (with b bad circuits up-front). For every ϵ , $c_0 = (T - BN)/T, \tau > 0$, if $N > i_0 / \left(\frac{B}{1-c_0} - \frac{B}{1-(1-\tau)c_0}\right)$ and $b > -(\lceil \log \epsilon \rceil + 1) / \log \left((1-\tau)c_0 + \frac{B}{B/(1-c_0)-i_0/N}\right)$, then $\text{Pr}_{fail}(N, B, T, \tau, b) < \epsilon$.*

Last, we also considered employing mixed strategies for BatchedCut protocols (i.e., fixing T to some public value up-front while randomizing the selection of B) to further reduce the cost. However, our analysis show that the extra improvement brought by randomized strategies is very small in this setting. This is consistent with our intuition: (1) It only makes sense to alternate B between two consecutive integers, which can be derived as a corollary of [12, Lemma 9]; (2) The strategy with smaller B is almost dominated by the one with larger B such that mixing them brings little extra benefit. Therefore, we opt to avoid using randomized strategies for BatchedCut protocols.

Results. Figure 8 depicts the improvements induced by the refined cost model for cut-and-choose. In this scenario, our search algorithm is able to identify the optimal pure strategies for r up to 10^5 , assuming the check rate c is always larger than 0.02. We note that the optimal strategy (characterized by (T, B) pairs) does not change much for $10^5 < r \ll \infty$. Experimental results show that roughly 20 ~ 80% performance gain can be achieved (while the exact improvement depends on r, ϵ and N). Note the effects of the bad circuits detection rate τ on the benefits of our approach (through comparing Figure 8a and 8b).

Changes to Existing Protocols. In this case, because we do not use randomized strategies, our proposal applied to the BatchedCut scenario requires *no protocol changes* other than setup the public parameters to the suggested value output by our search algorithm.

4 Benefits in Time

Recall that circuit checking will result in negligible network traffic because only a short circuit seed and a circuit

Input: ϵ, N, c_0 .

Output: T^*, B^*

1. Choose b^u and B^u .
 - (a) Let $s = \lceil \log \epsilon \rceil$. Compute $i_0 := s + 2$ and $b^u := -(s+1) / \log(\frac{c_0}{2} + \frac{2}{2/(1-c_0)-i_0/N})$
 - (b) Set $T^* := \infty$ and $B^u := \infty$.
2. For B ranging from 2 to B^u ,
 - (a) Precompute $\text{Pr}_e(N, B, b) = 1 - f_b(0)/f_b(1)$ for every $b \in [B, b^u]$.
 - (b) Find the smallest T , call it T_B , such that $T < T^*$ and $\text{Pr}_{fail}(N, B, T_B, \tau, b) < \epsilon$ for all $b \leq b^u$, using exponential backoff and binary search. (Note that $\text{Pr}_{fail}(N, B, T, \tau, b)$ monotonically decreases with T while the cost $T + NB(r-1)$ monotonically increases with T .)
 - (c) If $T^* + NB^*(r-1) > T_B + NB(r-1)$, then update $T^* := T_B$, $B^* := B$, and $B^u := \lceil (1 - c_0)T_B/N \rceil$.
3. Output T^*, B^* .

Figure 7: Find cost-efficient (T, B) for BatchedCut protocols.

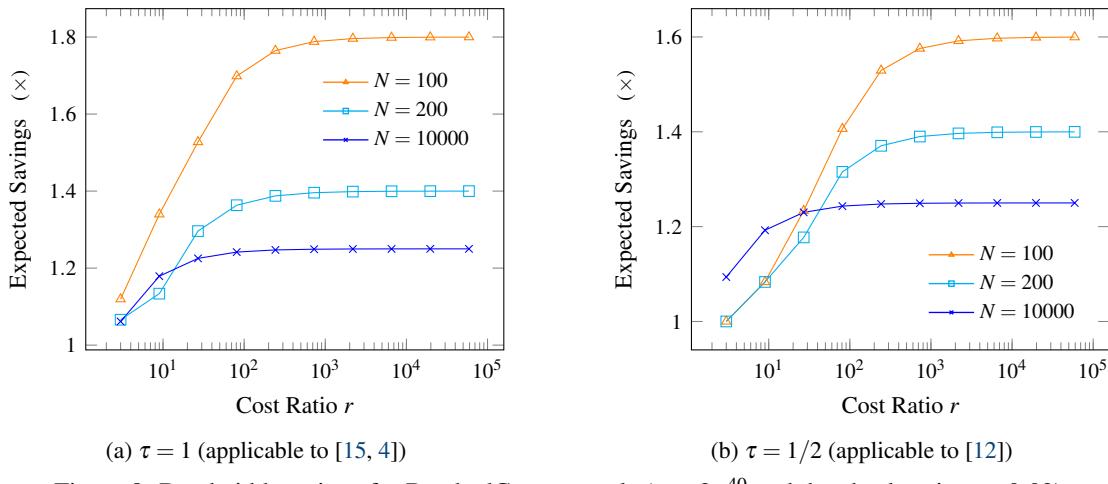


Figure 8: Bandwidth savings for BatchedCut protocols ($\epsilon = 2^{-40}$ and the check ratio $c \geq 0.02$)

hash needs to be transferred. This gap in bandwidth overhead also leads to a substantial gap in execution speeds, due to the significant difference in the throughputs of garbling/checking (about 50 ns/gate on a single-core processor) and that of network transmission (typically a wide range of 50~3000 ns/gate).

To evaluate the benefit of our technique in terms of time, we modified OblivC [29] to measure the ratios of speed for circuit evaluation and circuit checking tasks in various network settings. Our test implementation utilizes Intel AES-NI instructions and the half-gate garbling technique [30] to minimize bandwidth usage, and SHA256 implementation provided by Libgcrypt for circuit hashing. In circuit checking tasks, the circuit generator garbles a number of circuits but sends only a $(\text{Seed}, \text{Hash})$ pair for each garbled circuit, while the circuit verifier re-computes the hash from the seed for each circuit. We record the per circuit time cost for this task as T_c . For circuit evaluation tasks, the circuit generator garbles a number of circuits and sends the garbled gates to the evaluator, who not only evaluates, but also computes the hash of the received circuit. We record the per circuit time cost for this task as T_e . (In both tasks, the two

ends work in a pipelined fashion.) Thus, the time cost ratios between evaluation and checking can be calculated as $(T_c + T_e)/(2T_c)$ (recall that every circuits will be generated twice, once for committing their hashes and once for check/evaluate to avoid storing the typically gigantic circuits).

We used a benchmark circuit (provided by OblivC) with 31×10^6 non-free gates. Our experiments were run on two Amazon EC2 boxes (instance type: c4.large, \$0.105/hour, Intel Xeon E5-2666, 2.9GHz, 3.75GB memory) with Ubuntu 14.04 Server edition in the VA region.

We detail our experimental results in Table 5. The r values (in terms of wall-clock time) range from a little over 1 (with high speed connection) to 30 (with ordinary home-to-home connections). Such time gaps can be well-explained by the difference in the throughputs of computation and communication. The observed speedups of the proposed cut-and-choose technique can range from 12% up to 106%. Note that our approach yields no noticeable time savings for the settings of running SingleCut or BatchedCut protocols in a 1 Gbps LAN with single-core processors (compared to the their

state-of-the-art counterparts), because the cost ratio r is already very close to 1.

We note that due to the use of SHA256 in computing circuit hashes, we observe only 2.23×10^6 gates/second for circuit verification while 1.30×10^6 gates/second for circuit evaluation. It would be interesting to replace SHA256 with some hashing algorithm that leverages AES-NI instructions to match up with the speed of AES-NI based garbling (more than 10^9 gates/second, as was reported in [3]). That will imply a time ratio up to 100× larger than we observe in our experiments.

Table 5: Timing gaps between circuit evaluation and verification and our speedup benefits (measurements taken from 10 runs with 0.1% relative standard deviation) for a 31m gate circuit.

	LAN 1 Gbps	WAN 100 Mbps	WAN 10 Mbps
T_e (seconds)	24.1	103.5	818
T_c (seconds)	13.9	13.9	13.9
r	1.37	4.22	29.9
Speedup	MajorityCut	12%	26%
	SingleCut	0%	13%
	BatchedCut	0%	14%
			106%
			76%
			41%

5 Conclusion

The state-of-the-art design of cut-and-choose protocols considers an overly simplified cost model, and does not exploit the opportunity of dynamically variating e to thwart cheating adversaries. We have shown, through experiments, the dramatic gap in the bandwidth costs between circuit evaluation and circuit verification. We revisit the cut-and-choose protocol design problem in a refined cost model and give three highly efficient solvers, one for each class of cut-and-choose protocols, that output the best strategy for a particular cost ratio in our model. Simulation results show that our approach bring significant savings in bandwidth cost, as well as substantial speedups (especially when running secure computation protocols outside idealized laboratory environments). Most importantly, the benefits require very small changes to existing protocols and come completely from formal proofs that do not depend on any unprovable assumptions.

6 Acknowledgments

We thank Yuan Zhou (MIT) and Zhilei Xu (MIT) for inspiring discussions on cut-and-choose. We also appreciate Xiao Wang (Maryland)'s advice on benchmarking state-of-the-art garbled circuit implementations. Work of Ruiyu Zhu and Yan Huang was supported by NSF award #1464113. Work of Jonathan Katz was supported in part by NSF award #1111599. Work of shelat was supported in part by NSF grants TC-0939718, TC-1111781, and 1618559, a Microsoft Faculty Fellowship, and a Google Faculty Research Award.

Xiao Wang (Maryland)'s advice on benchmarking state-of-the-art garbled circuit implementations. Work of Ruiyu Zhu and Yan Huang was supported by NSF award #1464113. Work of Jonathan Katz was supported in part by NSF award #1111599. Work of shelat was supported in part by NSF grants TC-0939718, TC-1111781, and 1618559, a Microsoft Faculty Fellowship, and a Google Faculty Research Award.

References

- [1] A. Afshar, Z. Hu, P. Mohassel, and M. Rosulek. How to efficiently evaluate RAM programs with malicious security. *EUROCRYPT*, 2015.
- [2] A. Afshar, P. Mohassel, B. Pinkas, and B. Riva. Non-interactive secure computation based on cut-and-choose. *EUROCRYPT*, 2014.
- [3] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key block-cipher. *IEEE Symposium on Security and Privacy*, 2013.
- [4] L. T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique. *ASIACRYPT*, 2013.
- [5] N. Buescher and S. Katzenbeisser. Faster secure computation through automatic parallelization. In *USENIX Security Symposium*, Aug. 2015.
- [6] T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. *EUROCRYPT*, 2013.
- [7] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. *Cryptology EUROCRYPT*, 2008.
- [8] S. Gureron, Y. Lindell, A. Nof, and B. Pinkas. Fast garbling of circuits under standard assumptions. *Computer and Communications Security*, 2015.
- [9] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.
- [10] Y. Huang, J. Katz, and D. Evans. Efficient secure two-party computation using symmetric cut-and-choose. *CRYPTO*, 2013.
- [11] Y. Huang, J. Katz, V. Kolesnikov, R. Kumaresan, and A. J. Malozemoff. Amortizing garbled circuits. *CRYPTO*, 2014.
- [12] Y. Huang and R. Zhu. Revisiting LEGOs: Optimizations, analysis, and their limit. *Cryptology ePrint Archive*, Report 2015/1038, 2015. <http://eprint.iacr.org/2015/1038>.

- [13] N. Husted, S. Myers, A. Shelat, and P. Grubbs. Gpu and cpu parallelization of honest-but-curious secure two-party computation. *Annual Computer Security Applications Conference*, 2013.
- [14] B. Kreuter, A. Shelat, and C. hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*, 2012.
- [15] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. *CRYPTO*, 2013.
- [16] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. *EUROCRYPT*, 2007.
- [17] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 25(4):680–722, Oct. 2012.
- [18] Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. *Security in Communication Networks*, 2008.
- [19] Y. Lindell and B. Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. *CRYPTO*, 2014.
- [20] Y. Lindell and B. Riva. Blazing fast 2pc in the offline/online setting with security for malicious adversaries. *Computer and Communication Security*, 2015.
- [21] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy*, 2015.
- [22] P. Mohassel and B. Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. *CRYPTO*, 2013.
- [23] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel secure computation made easy. In *IEEE Symposium on Security and Privacy*, 2015.
- [24] J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. *Theory of Cryptography Conference*, 2009.
- [25] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. *ASIACRYPT*, 2009.
- [26] a. shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. *EUROCRYPT*, 2011.
- [27] a. shelat and C.-H. Shen. Fast two-party secure computation with minimal assumptions. *Computer and Communications Security*, 2013.
- [28] D. P. Woodruff. Revisiting the efficiency of malicious two-party computation. *EUROCRYPT*, 2007.
- [29] S. Zahur and D. Evans. Obliv-c: A language for extensible data-oblivious computation.
- [30] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. *EUROCRYPT*, 2015.
- [31] N. Smart. <https://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/>. Accessed on Feb 13, 2016.

A Proofs

Claim 1 Let T, B, N, b, i be defined as above. Then

$$\lim_{N \rightarrow \infty} \frac{\binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} = \left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i}\right)^{b-i}.$$

Proof There exists N_0 such that if $N > N_0$,

$$\begin{aligned} \frac{\binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} &= \frac{(T-b)!(T-BN)!(BN)!}{T!(T-BN-i)!(BN-b+i)!} \\ &= \frac{((T-BN-i+1)\cdots(T-BN))((BN-b+i+1)\cdots BN)}{(T-b+1)\cdots T} \\ &= \frac{(T-BN-i+1)\cdots(T-BN)}{(T-i+1)\cdots T} \cdot \frac{(BN-b+i+1)\cdots BN}{(T-b+1)\cdots(T-i)} \\ &\leq \left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i}\right)^{b-i} \triangleq U. \end{aligned}$$

Similarly, we have, there exists N_1 such that if $N > N_1$,

$$\frac{\binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \geq \left(\frac{T-BN-i+1}{T-i+1}\right)^i \left(\frac{BN-b+i+1}{T-b+1}\right)^{b-i} \triangleq L.$$

So, we know that, for sufficiently large N ,

$$\begin{aligned} U / \frac{\binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} &\leq \frac{U}{L} = \left(\frac{T-BN}{T-BN-i+1} \cdot \frac{T-i+1}{T}\right)^i \left(\frac{BN}{BN-b+i+1} \cdot \frac{T-b+1}{T-i}\right)^{b-i} \\ &= \left[\frac{(T-BN)T - (i-1)T + (i-1)BN}{(T-BN)T - (i-1)T} \right]^i \cdot \left[\frac{(BN-b+i+1)(T-i) + (b-i-1)(T-BN-i)}{(BN-b+i+1)(T-i)} \right]^{b-i} \\ &= \left[1 + \frac{(i-1)BN}{(T-BN)T - (i-1)T} \right]^i \left[1 + \frac{(b-i-1)(T-BN-i)}{(BN-b+i+1)(T-i)} \right]^{b-i} \\ &\leq \left(1 + \frac{i-1}{T-BN-i+1} \right)^i \left(1 + \frac{b-i-1}{BN-b+i+1} \right)^{b-i} \tag{3} \\ &\leq \left(1 + \frac{i-1}{T-BN-i+1} \right)^i \left(1 + \frac{b-1}{BN-b+1} \right)^b \tag{4} \end{aligned}$$

Note that the inequality (3) holds because $T > BN$. Thanks to the upper-bound of b (Claim 2) and hence on i (recall $i \leq b$), $\lim_{N \rightarrow \infty} \left(1 + \frac{i-1}{T-BN-i+1}\right)^i \left(1 + \frac{b-1}{BN-b+1}\right)^b = 1$. \blacksquare

Claim 2 Let $\Pr_{\text{fail}}(N, B, T, \tau, b)$ be the probability that the cut-and-choose game fails in a BatchedCut scheme (with b bad circuits up-front). For every ε , $c_0 = (T-BN)/T$, $\tau > 0$, if $N > i_0 / \left(\frac{B}{1-c_0} - \frac{B}{1-(1-\tau)c_0}\right)$ and $b > -(\lceil \log \varepsilon \rceil + 1) / \log \left((1-\tau)c_0 + \frac{B}{B/(1-c_0)-i_0/N}\right)$, then $\Pr_{\text{fail}}(N, B, T, \tau, b) < \varepsilon$.

Proof Let $0 < \tau \leq 1$ be the probability that P_2 detects the abnormality in checking garbled gate g conditioned on g is indeed bad. We have

$$\Pr_{\text{fail}}(N, b) = \sum_{i=0}^b (1-\tau)^i \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \Pr_e(N, b-i)$$

where $(1 - \tau)^i \binom{b}{i} \binom{T-b}{T-BN-i} / \binom{T}{T-BN}$ is the probability that P_1 who generates b bad gates survives the gate verification stage with i bad gates selected for verification (but P_2 fails to detect any of them). Because there exists i_0 such that $(1 - \tau)^{i_0} < \varepsilon/2$,

$$\begin{aligned}
\Pr_{fail}(N, b) &= \sum_{i=0}^b (1 - \tau)^i \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \Pr_e(N, b-i) \\
&= \sum_{i=0}^{i_0} (1 - \tau)^i \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \Pr_e(N, b-i) + \sum_{i=i_0+1}^b (1 - \tau)^i \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \Pr_e(N, b-i) \\
&\leq \sum_{i=0}^{i_0} (1 - \tau)^i \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \Pr_e(N, b-i) + (1 - \tau)^{i_0} \sum_{i=i_0+1}^b \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \Pr_e(N, b-i) \\
&\leq \sum_{i=0}^{i_0} (1 - \tau)^i \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \Pr_e(N, b-i) + \frac{\varepsilon}{2} \sum_{i=i_0+1}^b \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \\
&\leq \sum_{i=0}^{i_0} (1 - \tau)^i \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \Pr_e(N, b-i) + \frac{\varepsilon}{2} \sum_{i=1}^b \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \\
&\leq \sum_{i=0}^{i_0} (1 - \tau)^i \frac{\binom{b}{i} \binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \Pr_e(N, b-i) + \frac{\varepsilon}{2} \cdot 1 \\
&\leq \sum_{i=0}^{i_0} (1 - \tau)^i \binom{b}{i} \left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i}\right)^{b-i} \Pr_e(N, b-i) + \frac{\varepsilon}{2} \quad [\text{Claim 3}] \\
&\leq \sum_{i=0}^{i_0} (1 - \tau)^i \binom{b}{i} \left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i}\right)^{b-i} \Pr_e(N, b) + \frac{\varepsilon}{2} \\
&\leq \sum_{i=0}^{i_0} (1 - \tau)^i \binom{b}{i} \left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i_0}\right)^{b-i} \Pr_e(N, b) + \frac{\varepsilon}{2} \\
&\leq \sum_{i=0}^b (1 - \tau)^i \binom{b}{i} \left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i_0}\right)^{b-i} \Pr_e(N, b) + \frac{\varepsilon}{2} \\
&= \left((1 - \tau) \frac{T-BN}{T} + \frac{BN}{T-i_0}\right)^b \Pr_e(N, b) + \frac{\varepsilon}{2} \\
&\leq \left((1 - \tau) \frac{T-BN}{T} + \frac{BN}{T-i_0}\right)^b + \frac{\varepsilon}{2}.
\end{aligned}$$

Thus, $N > i_0 / \left(\frac{B}{1-c_0} - \frac{B}{1-(1-\tau)c_0} \right)$ ensures $(1 - \tau) \frac{T-BN}{T} + \frac{BN}{T-i_0} < 1$, while $b > -(s + 1) / \log \left((1 - \tau)c_0 + \frac{B}{B/(1-c_0)-i_0/N} \right)$ ensures $\left((1 - \tau) \frac{T-BN}{T} + \frac{BN}{T-i_0} \right)^b + \frac{\varepsilon}{2} \leq 2^{-s}$. Hence, we conclude that $\Pr_{fail}(N, B, T, \tau, b) < \varepsilon$. \blacksquare

Claim 3 If T, N, b, i are non-negative integers such that $T > BN$, $T \geq b$, and $i \leq b$, then

$$\frac{\binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} \leq \left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i}\right)^{b-i}.$$

Proof

$$\begin{aligned}
\frac{\binom{T-b}{T-BN-i}}{\binom{T}{T-BN}} &= \frac{(T-b)!(T-BN)!(BN)!}{T!(T-BN-i)!(BN-b+i)!} = \frac{[(T-BN-i+1) \cdots (T-BN)][(BN-b+i+1) \cdots BN]}{(T-b+1)(T-b+2) \cdots T} \\
&= \frac{(T-BN-i+1) \cdots (T-BN)}{(T-i+1) \cdots T} \cdot \frac{(BN-b+i+1) \cdots BN}{(T-b+1) \cdots (T-i)} \leq \left(\frac{T-BN}{T}\right)^i \left(\frac{BN}{T-i}\right)^{b-i}
\end{aligned}$$

On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis

Michael Backes

CISPA, Saarland University & MPI-SWS
Saarland Informatics Campus

Patrick McDaniel

Pennsylvania State University

Sven Bugiel

CISPA, Saarland University
Saarland Informatics Campus

Erik Derr

CISPA, Saarland University
Saarland Informatics Campus

Damien Ochteau

Pennsylvania State University &
University of Wisconsin

Sebastian Weisgerber

CISPA, Saarland University
Saarland Informatics Campus

Abstract

In contrast to the Android application layer, Android’s application framework’s internals and their influence on the platform security and user privacy are still largely a black box for us. In this paper, we establish a static runtime model of the application framework in order to study its internals and provide the first high-level classification of the framework’s protected resources. We thereby uncover design patterns that differ highly from the runtime model at the application layer. We demonstrate the benefits of our insights for security-focused analysis of the framework by re-visiting the important use-case of mapping Android permissions to framework/SDK API methods. We, in particular, present a novel mapping based on our findings that significantly improves on prior results in this area that were established based on insufficient knowledge about the framework’s internals. Moreover, we introduce the concept of permission locality to show that although framework services follow the principle of separation of duty, the accompanying permission checks to guard sensitive operations violate it.

1 Introduction

Android’s application framework—i.e., the middleware code that implements the bulk of the Android SDK on top of which Android apps are developed—is responsible for the enforcement of Android’s permission-based privilege model and as such is also a popular subject of recent research on security extensions to the Android OS. These extensions provide various security enhancements to Android’s security, ranging from improving protection of the user’s privacy [26, 46], to establishing domain isolation [29, 12], to enabling extensible access control [21, 8].

Android’s permission model and its security exten-

sions are currently designed and implemented as best-effort approaches. As such they have raised questions about the efficacy, consistency, or completeness [3] of the policy enforcement. Past research has shown that even the best-efforts of experienced researchers and developers working in this environment introduce potentially exploitable errors [15, 44, 35, 33]. In light of the framework size (i.e., millions of lines of code) and based on past experience [15, 44, 16, 33, 36], static analysis promises to be a suitable and effective approach to (help to) answer those questions and hence to demystify the application framework from a security perspective. Unfortunately, on Android, the technical peculiarities of the framework impinging on the analysis of the same have not been investigated enough. As a consequence, past attempts on analyzing the framework had to resort to simple static analysis techniques [7]—which we will show in this paper as being insufficient for precise results—or resort to heuristics [33].

In order to improve on this situation and to raise efficiency of static analysis of the Android application framework, one is confronted with open questions on *how to enable more precise static analysis of the framework’s codebase*: where to start the analysis (i.e., what is the publicly exposed functionality)? Where to end the analysis (i.e., what are the data and control flow sinks)? Are there particular design patterns of the framework runtime model that impede or prevent a static analysis? For the Android application layer, those questions have been addressed in a large body of literature. Thanks to those works, the community has a solid understanding of the sinks and sources of security- and privacy-critical flows within apps (e.g., well-known Android SDK methods) and a dedicated line of work further addressed various challenges that the Android *application* runtime model poses for precise analysis (e.g., inter-component communication [28, 40, 24, 27] or modelling the Android

app life-cycle[25, 6]). Together those results form a strong foundation on which effective security- and privacy-oriented analysis is built upon. In contrast to the app layer, for the application framework we have an intuitive understanding of what constitutes its entry points, but no in-depth technical knowledge has been established on the runtime model, and almost no insights exist on what forms the security and privacy relevant targets of those flows (i.e., what technically forms the sinks or “protected resources”).

Our Contributions. This paper contributes to the demystification of the application framework from a security perspective by addressing technical questions of the underlying problem on *how* to statically analyze the framework’s code base. Similar to the development of application layer analyses, we envision that our results contribute some of the first results to a growing knowledge base that helps future analyses to gain a deeper understanding of the application framework and its security challenges.

How to statically analyze the application framework. We present a systematic top-down approach, starting at the framework’s entry points, that establishes knowledge and solutions about analyzing the control and data flows within the framework and that makes a first technical classification of the security and privacy relevant targets (or resources) of those flows. The task of establishing a precise static runtime model of the framework was impeded by the absence of any prior knowledge about framework internals beyond black-box observations at the framework’s documented API and manual analysis of code fragments. Hence we generate this model from scratch by leveraging existing results on statically analyzing Android’s application layer at the framework layer. The major conceptual problem was that the design patterns of the framework strongly differ from the patterns that had been previously encountered and studied at the application layer. Consequently we devised a static analysis approach that systematically encompasses all framework peculiarities while maintaining a reasonable runtime. As result of this overall process, we have established a dedicated knowledge base that subsequent analyses involving the application framework can be soundly based upon.

AXPLORER tool and evaluation. Unifying the lessons learned above, we have built an Android application framework analysis tool, called AXPLORER. We evaluate AXPLORER on four different Android versions—v4.1.1 (API level 16), v4.2.2 (17), v4.4.4 (19), and v5.1 (22)—validate our new insights and demonstrate how specialized framework analyses, such as message-

based IPC analysis and framework component interconnection analysis, can be used to speed up subsequent analysis runs (e.g. security analyses) by 75% without having to sacrifice precision. As additional benefit the resulting output can be used by independent work *as is* to create a precise static runtime model of the framework without the need to re-implement the complex IPC analysis.

Android permission analysis. Finally, to demonstrate the benefits of our insights for security analysis of the framework, we conduct an Android permission analysis. In particular, we re-visit the challenge of creating a permission map for the framework/SDK API. In the past, this problem has been tackled [32, 7] without our new insights in the peculiarities of the framework runtime model, and our re-evaluation of the framework permission map reveals discrepancies that call the validity of prior results into question. Using AXPLORER, we create a new permission map that improves upon related work in terms of precision. Moreover, we introduce a new aspect of permission analysis, permission locality, by investigating which framework components enforce a particular permission. We found permissions that are checked in up to 10 distinct and not necessarily closely related components. This indicates a violation of the separation of duty principle and can impede a comprehensive understanding of the permission enforcement.

2 Background

We first provide necessary technical background information on the Android software stack and the abstract control and data flows in the system. Android OS is an open source software stack on top of the Linux OS. Between the apps at the top of the stack and the Linux kernel at the bottom is the Android middleware. This middleware consists of the application runtime environment, default native libraries (like SSL), and the Java-based application framework (see Figure 1).

2.1 Android Application Framework

The application framework consists of the various services that implement the Android app API (e.g., retrieving location data or telephony functionality). Every framework service is responsible for providing access to one specific system resource, such as geolocation, radio interface, etc.

Bound services. These services are implemented as bound services [4] as part of the SystemServer.

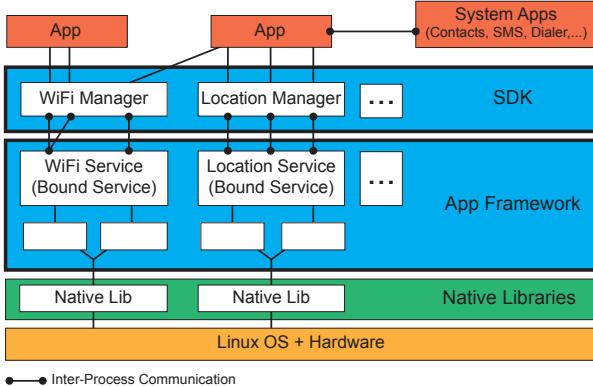


Figure 1: Android Software Stack with abstract control and data flows.

Bound service is the fundamental pattern to realize Android services that are remotely callable via a well-defined interface. Such interfaces are described in the *Android Interface Definition Language* (AIDL) and an AIDL compiler allows automated generation of **Stub** and **Proxy** classes that implement the interface-specific Binder-based RPC protocol to call the service. Here, **Stubs** are an abstract class that implements the **Binder** interface and needs to be extended by the actual service implementation. **Proxies** are used by clients to call the service. On top of **Stubs** and **Proxies**, the Android SDK provides **Managers** as abstraction from the low-level RPC protocol. **Manager** classes encapsulate pre-compiled **Proxies** and allow developers to work with **Manager** objects that translate local method calls into remote-procedure calls to their associated service and hence enable app developers to easily engage into RPC with the framework’s services. However, **Proxies** and **Managers** are just abstractions for the sake of app developer’s convenience and nothing prevents an app developer from bypassing the **Managers** or re-implementing the default RPC protocol to directly communicate with the services.

A small number of framework services does not use AIDL to auto-generate their **Stub/Proxy**, but instead provides a custom class that implements the **Binder** interface. The most prominent exception is the **ActivityManagerService** (AMS), which provides essential services such as application life-cycle management or Intent distribution. Since its interface is also called from native code, for which the AIDL compiler does not auto-generate native **Proxies/Stubs** and hence requires manual implementation of those, the RPC protocol for the AMS is hardcoded to avoid misalignment between manually written and auto-generated code.

The services are an essential part of the middleware front-end to the application layer and calling their interfaces triggers control and data flows within the application framework. Naturally, the flows of some services lead to interaction with the underlying platform through the native libs. For instance, the **WifiService** is interacting with the WiFi daemon. Other services, such as Clipboard, do not rely on any hardware features. However, the exact control and data flows have not yet been studied or charted (see blank boxes in Figure 1) and facilitating this mapping by enabling analysis of the framework is part of the contributions of this work.

System apps. System apps, such as Contacts, Dialer, or SMS complement the application framework with commonly requested functionality. However, in contrast to the application framework services that are fixed parts of any Android deployment, system apps are exchangeable or omittable (as can be observed in the various vendor customized firmwares) and, more importantly, are simply apps that are programmed against the same application framework API as third-party applications.

2.2 Permissions

One cornerstone of the Android security design are *permissions*, which an app must hold to successfully access the security and privacy critical methods of the application framework. Every application on Android executes under a distinct Linux UID and permissions are simply Strings¹ that are associated with the application’s UID. There is no centralized policy for checking permissions on calls to the framework API. Instead, framework services that provide security or privacy critical methods to applications (must) enforce the corresponding, hard-coded permission that is associated with the system resources that the services expose. To enforce permissions, the services programmatically query the system whether their currently calling app—identified by its UID—holds the required permission, and if not take appropriate actions (such as throwing an exception). For instance, the **LocationManagerService** would query the system whether a calling UID is associated with the String `android.permission.ACCESS_FINE_LOCATION`, which represents the permission to retrieve the GPS location data from the **LocationManagerService**.

In this model, system apps differ from third-party apps in that they can successfully request security

¹Permissions that map to Linux GIDs do not involve the framework and are not further considered here.

and privacy critical system permissions from the framework, which are not available to non-system apps. Moreover, like framework services (and any non-system application), they are responsible for enforcing permissions for resources they manage and expose on their RPC interfaces (e.g., contacts information or initiating phone calls). The difference to non-system applications is, that they usually enforce well-known permissions defined in the Android SDK, although the Android design does—in contrast to the framework services—not hardcode where those permissions are enforced, thus allowing system apps to be exchanged.

3 Related work

Static (app) analysis. Different related works have analyzed Android apps for vulnerabilities and privacy violations. Enabling precise static app analysis required solving essential questions like what are the entry points of the app, what are the security relevant sinks and how can we achieve a static runtime model that takes the application peculiarities into account? Among the static analysis approaches, *CHEX* [25] was the first tool to accommodate for Android’s event-driven app lifecycle with an arbitrary number of entry points. *FlowDroid* [6] further improved the runtime model by automatically generating per-component lifecycle models that take into account the partial entry point ordering. While *FlowDroid* still analyzed components in isolation, a number of related works specifically addressed the problem of inter-component communication (ICC). The initial work *Epicc* [28] devised a new analysis technique to create specifications for each ICC sink and source. *Amandroid* [40] combined a lifecycle-aware program dependence graph with ICC analysis to generate an inter-component model of the application to improve precision for various security applications. Similarly, *IccTA* [24] extended *FlowDroid* with a precise inter-component model. Finally, *IC3* [27] uses composite constant propagation to improve retargeting of ICC-related parameters enabling a more precise ICC resolution. Moving from best effort approaches, *SuSi* [5] took a machine-learning approach for classifying and categorizing sources and sinks in the framework code that are relevant for application analysis. All of those solutions contribute to analyzing Android apps more efficiently. The focus of this work is on establishing similar knowledge on Android’s application framework and on making a first essential but non-trivial step towards enabling a holistic analysis of Android that includes the framework code with its security architecture.

Application Framework Abstractions. The application framework is generally regarded as too complex to be considered in an app analysis (cf., *CHEX* [25]) and very recent works dealt specifically with this problem of abstracting the application framework [13, 18] or making it amenable for app analysis [11]. *EdgeMiner* [13] links callback methods to their registration methods and generates API summaries that describe implicit control flow transitions through the framework. *DroidSafe* [18] distills a compact, data-dependency-aware model of the Android app API and runtime from the original framework code. *Droidele* [11] differs in its approach by explicating the reflective bridge between the application framework and applications, while trying to model the framework as less as possible. It generates app-specific versions of the application framework and replaces reflective calls with app-specific stubs. All of these approaches try to pre-compute data-dependencies through the framework API that can be used by app analyses in favor of using the complex and huge framework code base. In contrast, our work makes a first step towards enabling in-depth analyses of the application framework beyond just data dependencies in order to enable future reasoning about framework security architectures or extensions (such as guiding and verifying hook placement or separation of duties).

Permission Mapping and Inconsistencies. Both *Stowaway* [32] and *PScout* [7] built permission maps for the framework API. Stowaway used unit testing and feedback directed API fuzzing of the framework API to observe the required permission(s) for each API call. PScout, in contrast, used static reachability analysis between permission checks and API calls to create a permission mapping of different Android framework versions that improves on the results of Stowaway. Permission maps have since been a valuable input to different Android security research, such as permission analysis [20] and compartmentalization [31, 34] of third-party code, studying app developer behavior [32, 38], detecting component hijacking [25], IRM [23, 10] and app virtualization [9], or risk assessment [30, 19, 45, 42]. In this work, we re-visit the challenge of creating a permission map for Android. In contrast to the prior work, we build on top of our new insights on how to statically analyze the application framework (see Sections 4 and 5), which allow us to achieve a map that is more precise for the application framework API and that calls the validity of some prior results [7] into question. We discuss how recent work [33] that focused on inconsistent security enforcement within the framework

could benefit from a deeper understanding of the framework’s peculiarities separately in Section 8.

Android Security Frameworks. Various security extensions have been proposed, such as [26, 46, 29, 12, 21, 8] to name a few, which integrate authorization hooks into Android’s application framework to enforce a broad range of security policies. At the moment, those extensions are designed and implemented as best-effort approaches that raise questions about the completeness and consistency of the enforcement and indeed past research has shown that even the best-efforts of highly experienced researchers and developers working in this environment introduce potentially exploitable errors [15, 44, 35, 33]. This unsatisfying situation has strong parallels to earlier work on integrating authorization hooks into the Linux and BSD kernels [41, 39], where a dedicated line of work [15, 44, 16] has established tools and techniques to reason about the security properties of proposed extensions or to automate the hook placement. Prerequisite for those solutions was a clear understanding of what constitutes a resource that is (or should be) protected by an authorization hook. To allow development of similar tools for the Android application framework, we hence have to also answer the question about Android’s protected resources first. In this work we want to make a first essential step in this direction by enabling a deeper analysis of the framework and by providing a first high-level taxonomy of protected resources in the application framework.

4 Enabling In-Depth Application Framework Analysis

In contrast to the various related works on static analysis at the application level, there is no existing prior work on in-depth analysis of the application framework. Moreover, as the architecture of the framework fundamentally differs from the architecture of applications, open questions have to be answered first to be able to conduct in-depth static analysis of the framework. For instance, “*what are the entry points to the application framework?*” or “*how to establish a static runtime model of the framework’s control flows?*” In the following we identify challenges that arise for static analyses at framework level and present a systematic, top-down approach to cope with these problems (an implementation of our approach is presented in Section 5). Solving the discussed challenges lays the foundation on which a wide range of security analyses of the application frame-

work can be constructed, from which we (re-)visit the use-case of permission analysis in Section 7.

4.1 Defining Framework Entry Points

The first question to be answered is how to identify and select the starting points for the framework analysis? At application level this has already been studied in depth [25, 6, 14, 17, 43]. From a high-level view, most approaches parse the declared components from the application manifest and determine the components as well as dynamically registered callbacks as entry points; or they build component/application life-cycle models with a single main entry method.

Challenge: *The framework model is conceptually different from the application layer and existing approaches for application layer analysis do not apply in a framework analysis. Instead one has to identify the framework API methods that are exposed to app developers as analysis entry points.*

To identify the entry point methods, we have to locate the relevant framework entry point classes. Starting with the official API of the Android SDK (e.g., `Managers` in Figure 1) is not reliable as there are no means to prevent an app developer from bypassing the SDK by immediately communicating with the framework services or by using reflection to access hidden API methods of the SDK. Consequently, we do not consider the API calls within the SDK as entry points but instead the framework classes that are entry points for accessing framework functionality (i.e., framework classes that are being called by the SDK, see Figure 1). We exclude entry points that are not accessible by app developers, such as Zygote, service manager, or the property service, which are under special protection (e.g., SELinux [37]) and will not accept commands by third-party apps that have tangible side-effects on the system or other apps. This restriction is in accordance with the design of existing Android security extensions, which exclusively focus on the exported functionality of the app framework (e.g., framework’s bound services).

Inter-component communication in Android is by design based on `Binder` IPC and, thus, framework classes have to expose functionality via `Binder` interfaces to the application layer. To this end, interfaces must be derived from `IInterface`, the base class for `Binder` interfaces. `Binder` interfaces might be automatically generated by AIDL, in this case the entry point classes extend the auto-generated `Stub` class, or in case of `Binder` interfaces that are not generated by AIDL, a custom `Binder` implementation

like `ActivityManagerNative`² has to be provided, which in turn is extended by the entry point classes. These class relationships can be resolved via a class hierarchy analysis (CHA) to determine the set of all entry classes. Besides bound services this also includes callback and event listener classes that expose an implementable interface to app developers. Hence, we define entry points (EP) as the public methods of framework classes that are exposed via a `Binder` interface. In addition, permission-protected entry points (PPEP) are defined as entry points from which a permission check is control-flow reachable.

4.2 Building a Static Runtime Model

Challenge: *Generating a static model that approximates the runtime behaviour of the application framework again strongly differs from the problems that arise at application level where the component life-cycles are mimicked to approximate runtime behavior. The bound services—as entry points to the framework—might be queried simultaneously from multiple clients (apps) via IPC and hence have to handle multi-threading to ensure responsiveness of the framework. In contrast to the application space at which utility classes like `AsyncTask` are used for threading, we discovered that the framework services make intensive use of more generic but also more complex threading mechanisms like `Handler`, `AsyncChannel`, and `StateMachines`. Disregarding these concurrency patterns results in imprecise data models that cause a high number of false positives during framework analysis.*

In the following, we provide technical background for those asynchronicity patterns and explain how they can modeled correctly for static analyses.

Handler. The class `android.os.Handler` provides a mechanism for reacting to messages or submitting Java `Runnable` objects for execution on a (potentially remote) thread. `Handlers` either schedule the processing of a message or the execution of a `Runnable` at some point in the future or process a message/`Runnable` on a separate thread.

To illustrate the `Handler` mechanism, consider the example shown in Listing 1. It includes the relevant sections of the framework class `com.android.server.BluetoothManagerService`. When the service is constructed, it instantiates a `HandlerThread` object (line 6), a traditional `Thread` object associated with a `Looper`. The

```

1| class BluetoothManagerService {
2|     private HandlerThread mThread;
3|     private BluetoothHandler mHandler;
4|
5|     public BluetoothManagerService() {
6|         mThread = new
7|             HandlerThread("BluetoothManager");
8|         mThread.start();
9|         mHandler = new
10|             BluetoothHandler(mThread.getLooper());
11|     }
12|     public void enable() {
13|         Message msg =
14|             mHandler.obtainMessage(MESSAGE_ENABLE);
15|         mHandler.sendMessage(msg);
16|     }
17|     public void disable() {
18|         Message msg =
19|             mHandler.obtainMessage(MESSAGE_DISABLE);
20|         mHandler.sendMessage(msg);
21|     }
22|
23|     class BluetoothHandler extends Handler {
24|         public void handleMessage(Message msg) {
25|             switch (msg.what) {
26|                 case MESSAGE_ENABLE:
27|                     // process enable message
28|                     break;
29|                 case MESSAGE_DISABLE:
30|                     // process disable message
31|                     break;
32|                 // Other cases.
33|             }
34|         }
35|     }
36| }
```

Listing 1: Bluetooth Handler in the Bluetooth manager service. Code was simplified for readability.

purpose of the `Looper` class is to sequentially process the messages in a message queue. At line 8, the class-specific `BluetoothHandler` object is created and associated with the newly created `Looper` from the `HandlerThread`. This allows messages sent to the `BluetoothHandler` to be pushed to the message queue for this `Looper`. Methods `enable` and `disable` can be called by applications via RPC on `IBluetoothManager` to turn the bluetooth functionality on or off. Method `enable` sends a message with code `MESSAGE_ENABLE` to the `BluetoothHandler` (line 12). When the associated `Looper` instance processes the message, it calls method `handleMessage` in the `BluetoothHandler` (line 20), which then processes the request.

Statically resolving message-based IPC, requires to overcome several challenges. First, the target `Handler` type has to be inferred, to determine the concrete `handleMessage` method of the receiving class that processes the message. Second, to add precision to the analysis, it is best to make it locally path-sensitive by inferring the possible message codes of the arguments to `sendMessage` methods. For the example presented in Listing 1, this enables the analysis to be limited to the feasible paths for a given message in the switch at line 21. While it is possible to per-

²By convention non-generated class names end with `Native`.

form the analysis without this information, doing so results in a significant loss of precision and, thus, an increase in the number of false positives, which may distort the results of security analyses built on top. In light of the prevalence of the `Handler` pattern, this loss of precision is not an acceptable solution. Finally, since messages can also be associated with runnable tasks instead of message codes, the concrete `Runnable` types associated with each message have to be inferred to determine the runnable code executed when such a message is processed.

AsyncChannel. Closely related to `Handlers`, `com.android.internal.util.AsyncChannel` implements a bi-directional channel between two `Handler` objects. It provides its own `sendMessage` and `replyToMessage` methods, both of which delegate to the `sendMessage` methods in its associated `Handler`. In order to precisely model `AsyncChannel` objects, it is necessary to infer the types of the sender/receiver `Handler` objects. Similarly to `Handlers`, path-sensitivity should be added to the analysis by inferring the message codes that are sent through the channel.

StateMachine. Building on the `Handler` concept, the `com.android.internal.util.StateMachine` class models complex subsystems such as the DHCP client or the WiFi connectivity manager. This class allows processing of messages depending on the current state of the modeled system. It effectively constitutes a hierarchical state machine in which messages cause state transitions. States are organized in a hierarchical manner, such that parent states may process messages that are not handled by child states. In order to precisely model state machines, several challenges must be addressed. First, the subtype of the state machine itself must be inferred, with all the states and possible transitions. Second, the hierarchy of the states must be inferred, in order to know which `enter` and `exit` state methods are called upon state transitions. Moreover, this is necessary to know which state may handle a given message. Third, for eliminating further false positives one needs to infer the possible states for any given program location at which interaction with the state machine occurs.

4.3 Identifying Protected Resources

While the previous sections show how static analysis of the android application framework code base can be enabled, we now classify the resources inside the application framework that actually have

to be protected. Unfortunately, there is a lack of consensus in the community on what constitutes a security-sensitive resource/operation [5, 16] and no one-size-fits-all definition exists as the concrete definition depends on various aspects like operating system, programming language, or even the domain. To avoid ambiguities on what we denote as protected resource in the remainder of this paper, we note that protected resources for us are security sensitive operations that have a tangible side effect on the system state or use of privacy.

Challenge: *Defining the security-relevant resources is, in contrast to entry points, more challenging. For privacy leak analysis at application-level, there is a well-defined list of API methods that can be classified as sinks. Since the analysis now shifts into the API methods of the framework, it is unclear what kind of resources are protected by Android’s permissions and can, thus, be used as sinks for security analysis within the framework.*

To create a first high-level taxonomy of protected resources that can help to automatically discover such resources, we first have to create a ground truth about what technically forms a protected resource. To this end, we manually investigated control flows of a number of identified PPEP in the framework’s source code. Here, we make the assumption that every existing permission check within the application framework indeed controls access to at least one security- or privacy-critical system resource. Checks are usually located at the very beginning of PPEP, so that any subsequent operation is indeed authorized. Using expert knowledge in combination with descriptions of expected side effects from the Android documentation we identify and annotate relevant statements that modify the service and/or system state. To avoid a potential bias in the types of protected resources, we chose entry points from eight different entry classes. To cover a variety of disjunct cases, we based our selection on the available information such as return value, number/type of EP input arguments, or number/type of permission checks collected during the entry point discovery. After manually investigating flows from 35 entry points, distinct repetitive patterns for protected resources appeared across the different control flows, which we summarized in a taxonomy of the high-level protected resource types.

Taxonomy of protected resources. Figure 2 presents our high-level taxonomy of the protected resource types. In contrast to work at application-level that disregards field instructions [5], we found that `field update` instructions are highly relevant in the

context of the framework and in fact are the most prevalent type of protected resources that we discovered. Relevant *method invocations* can be further sub-classified into native method calls (e.g. for file system access or modification of device audio settings) and broadcast sender. We consider native method calls generally as protected resources, since distinguishing non-/security-relevant native calls would require a dedicated analysis for the native code, which is currently a general, open problem for the community and out of scope for this work. Broadcast senders are protected resources as they can potentially cause side-effects on the system or apps. However, this is statically unresolvable, as the concrete side-effects strongly depend on the current system configuration, e.g. on the installed apps and the set of active broadcast listener. We consider non-void *return values* of security-sensitive entry methods as protected resource. Returned objects of such methods constitute sensitive data, e.g., a list of WiFi connections. Return values of primitive types `int` or `boolean` may constitute sensitive values like for the method `isMultiCastEnabled` of the `WifiService` or some status/error code in method `enableNetwork` of the same service. We also found cases in which a *throw RuntimeException* (RTE) has to be considered as a protected resource. For instance, in the `crash` method of the `PowerManagerService`, which requires from the caller the permission to reboot the device, an RTE causes the runtime to crash and the device to reboot in consequence.

Coverage of the taxonomy. An inherent limitation of our taxonomy based on small-scaling manual analysis is, that there are no guarantees that corner cases are included in the current classification. To cover all corner cases in our taxonomy, a comprehensive manual analysis of the framework would be required, which would defeat the purpose of enabling a static analysis in the first place. This constitutes a high-level taxonomy of protected resource (types) in the framework. Distilling a more refined set for security analyses is discussed separately in Section 8.

5 Implementation

We combined all aforementioned steps from Section 4 for analysis of an arbitrary framework version into a tool called `AXPLORER`. We leverage the static analysis framework `WALA` [2], although our approach is equally applicable to other analysis frameworks such as `Soot` [1]. Additional code for realizing our approach comprises ≈ 15 kLOC of Java.

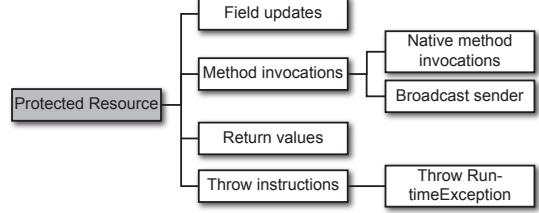


Figure 2: High-level taxonomy of protected resource operation types.

Call-graph generation. For each identified entry class, we generate an inter-procedural call-graph (CG). As opposed to related approaches [7] that use class hierarchy analysis to generate low-precision call-graphs due to the overall framework complexity—Android version 4.2.2 already includes over 35,000 classes—we generate high-precision call-graphs with object-sensitive pointer resolution. For each virtual or interface invocation we infer the runtime type(s) and hence precisely connect the invocation to its target(s). Although the costs for the points-to computation are computational very expensive, the increased precision lowers the complexity of the overall call-graph, since we do not introduce imprecision by considering all subclasses of a virtual method call as potential receivers. Avoiding this imprecision in the call-graph also lowers the number of false positives. The complexity is further reduced by the design decision to not follow RPC calls to other entry classes. We complement the call-graph with message-based IPC edges during the control-flow slicing (see below).

Slicing & on-demand msg-based IPC resolution. We conduct a forward control-flow slice for each identified entry point method. The slicer stops at native methods, RPC invocations to classes other than the current one, and when the entry point method returns. During slicing, we perform an on-demand message/handler resolution to add message-based IPC edges to the call-graph, thus avoiding a huge computational overhead of computing all edges in advance when only a subset of them are required for analysis (e.g. if PPEP are analyzed only).

When the slicer reaches a `sendMessage` call, we infer the concrete handler type and add a call edge from the `sendMessage` call to the `handleMessage` method of the receiving handler. We augment this process with inter-procedural backwards slicing for two reasons: First, since existing type inference algorithms (like the ones implemented in `WALA`) work intra-procedurally, type inference fails if `Handler` objects are stored in fields whose declared field type is the

`Handler` base class and not the concrete subtype. Using inter-procedural backwards slicing starting at the message-sending instruction, we obtain a more precise set of possible handler types in AXPLORER. Second, `Messages` are usually not constructed explicitly but indirectly obtained via calls to `Message.obtain` or `Handler.obtainMessage` and contain a public integer field that carries a sender-defined message code that allows the recipient to identify the message type. To statically identify the message code we compute a backwards slice starting from the message-sending instruction and check the resulting set of instructions for calls that construct/obtain a message. We then repeat this approach starting from the message obtain call to infer the concrete message code used to initialize the `Message`.

`Handlers` use `switch` statements to match the provided message code and to transfer control-flow to a specific basic block of the method’s control-flow graph (cf. line 25 et seqq. in Listing 1). To avoid infeasible paths, we have to recreate path-sensitivity intra-procedurally and map the message code(s) to the individual execution path(s). The control-flow slicer then continues at this specific execution path to avoiding a huge number of false positives. `Runnable` types on a `post` call of the `Handler` are resolved in the same way and a call edge to the `Runnable`’s `run` method is added. The approach slightly differs in case of `StateMachines`. Here, there is no single `handleMessage` function. Instead, each `StateMachine` implements its own `processMessage` function. In this case, we recreate path-sensitivity for each of these functions and delegate the control-flow to any matching switch statement.

6 Framework Complexity Analysis

We apply our gained insights from Section 4 to collect complexity information about the application framework. By doing this, we demonstrate how the analysis complexity can be held manageable to allow such in-depth analysis within a reasonable amount of time. Finally, we collect the framework’s protected resources as denoted in our taxonomy and validate the results (a detailed discussion on how security analyses can benefit from this is given in Section 8). Using AXPLORER we analyze four different Android versions: 4.1.1 (API level 16), 4.2.2 (17), 4.4.4 (19), and the latest Lollipop release 5.1 (22).

6.1 Handling Framework Complexity

Table 1 summarizes different complexity statistics generated for the four analyzed versions. Unsur-

prisingly, the complexity in terms of code increases with each version, whereas the gap to the most recent major version is significantly larger as between the minor version changes due to new features like Android TV. The entry class discovery algorithm identified between 242–383 entry classes of which $\approx 25\%$ include at least one PPEP. The evaluation was conducted on a server with four Intel Xeon E5-4650L 2.60 GHz processors with 8 cores each and 768 GB RAM. Initial processing of the frameworks finished in reasonable time, ranging from 14–126 hours. Note that this computation has to be done only once per Android version and that there are no real-time constraints as, e.g., in application vetting. The most time-consuming task (about 85% of the overall time) was generation of the high-precision call-graphs. In the following, we describe the use of entry-class interconnection and IPC analysis to speed up processing time without loosing the precision of our data model.

Entry class interconnection. IPC-interfaces of framework entry classes are not only used by the application layer, but also by other framework services. Analyzing the communication behavior of entry classes does not only provide a deeper understanding of how the framework services are interconnected but also facilitates analyses that rely on permission checks as security indicator (e.g., see Section 7). Exploiting the knowledge about which service EP triggers which RPCs along its control flow enables pre-computation of execution path conditions and restricting the scope of a service analysis to only subsets of dependent services rather than the entire framework (i.e, it allows to efficiently divide and conquer the framework analysis). In a post-processing step the analysis results for distinct services can be stitched together at RPC boundaries. Appendix A illustrates the RPC interconnections for Android 5.1.

Message-based IPC Analysis. A precise model of the message sender to handler relations is crucial for the generation of a static runtime model of the framework with a low number of false connections. The last row in Table 1 shows the prevalence of the message sending pattern. Between 38–52% of PPEP include at least one message sending call. Across API levels we found 300 (API 16) to over 500 (API 22) distinct message sender calls used within PPEP. The evaluation of our IPC analysis showed that in 7% of all cases the message was sent to a `StateMachine`, and in 27% of all cases to a `Handler`. In the remaining 66% a `Runnable` was posted. This ratio remains approximately the same in all versions. Overall, our IPC analysis was able to fully resolve about 76%

Android version	4.1.1 (16)	4.2.2 (17)	4.4.4 (19)	5.1 (22)
# of classes	27,749	29,804	31,023	46,192
- inner classes	14,784	15,936	17,525	28,933
# of entry point classes	242	256	284	383
- with at least one PPEP	64 (26.4%)	73 (28.5%)	75 (26.4%)	81 (21.2%)
# entry methods (EP)	2,583	2,734	2,861	3,225
- with perm check (PPEP)	863 (33.4%)	1,018 (37.2%)	1,227 (42.9%)	1,250 (38.8%)
- incl. message sending	328 (38.0%)	532 (52.2%)	518 (42.2%)	597 (47.8%)

Table 1: Comparing complexity measures for different Android versions (percentages relate to preceding line).

of all message sending instances, yielding already a very valuable data set of the message sender to handler relationships. Reasons for failed resolution are that either the `Handler` (81%) or `Runnable` (5%) could not correctly be inferred while in the remaining 13% of cases the message code could not be inferred. The root cause of most of these failures is the missing/incomplete support of `AsyncChannels` and the `Message.sendToTarget()` API call. At the time of writing this support is work-in-progress.

During our initial analysis run AXPLORER records both an RPC-map per entry class as well as a list of resolved sender-to-handler relationships. This data is then re-applied as expert knowledge in subsequent analysis *re-runs* to significantly reduce the analysis runtime, e.g., for API level 17, the processing time drops by ~75% to about 7 hours. By publishing this data we hope that independent analyses can equally benefit from this by removing the burden to re-implement a comparable IPC resolution algorithm.

Reflection We analyzed reflection usage within framework code by counting the number of calls to methods within the `java.lang.reflect` package. The absolute numbers range from 89 (API 16) to 118 (API 22). Across API levels less than 50% targeted the `Method` class while the remaining calls were distributed among other reflection classes. In many cases reflection is used in utility or debug classes and we found only one entry class that makes use of reflection (`ConnectivityService`), but the respective method was removed in API level 20. In SDK code the total numbers are slightly higher across API levels (115–288). However, the additional usage of reflection is mainly due to `View/Widget` classes. Overall, reflection is only rarely used in framework code and not used at all by main service components.

6.2 Android’s Protected Resources

To validate our established taxonomy, we collect the protected resources for each Android version and classify them with respect to the taxonomy. Across

versions the total number ranges from 6,5k (API 16) to 10k (API 22). Although these numbers seem quite high at first glance, they are reasonable in relation to the overall size and complexity of the framework. AXPLORER recorded the context depth (in terms of method invocations) at which the protected resources were found. While for simple methods that include few (or only even one) resource the call depth is lower than two, the median call depth ranges from 8–11 across Android versions. This emphasizes that approaches that do not perform in-depth analysis are not suitable to detect resources located deeper in the control-flow. The relative distribution of resources per type is stable across all versions. We validated our statement of Section 4.3 that field update instructions are the most prevalent resource type (with a share of about 75%). They are followed by native method calls (about 21–23%), which are most frequently used as a gateway to the device hardware (e.g. file system, audio, nfc). There is a surprisingly low number of PPEP that return a protected value, the absolute number ranges from 51–69 entries. Another unexpected result is that runtime exceptions occur with a frequency that is about as high as protected broadcast senders. Besides the already mentioned example within the `PowerManagerService`, we found occurrences in UI widget classes and even in the default XML parsing library on Android.

Appendix B gives more detailed statistics on protected resources, as well as a manual validation and assessment of the use of RTE in the framework.

7 Permission Analysis

Building on top of our new insights we re-visit an important aspect of Android’s permission specification, that is *permission mapping* between permission check and SDK method, and further introduce *permission locality* to study which framework components perform which permission checks. To this end we extend AXPLORER as follows:

- 1) A PPEP only indicates the presence of a permission check in the control-flow from this

entry-point, but there is no information yet about the number of checks or the concrete permission strings. We extend our slicing-based approach to also resolve the permission strings in common permission check API invocations (e.g., as defined in the `Context` class). Non-constant strings are resolved in a similar way like message codes in Section 5. From 520 distinct permission checks found in API level 16, we were able to resolve 99% of the permission strings. Among the failing cases, one case was located in the `ActivityManagerService$PermissionController` class where the permission string is an argument of the entry point method, which is only called from native code and hence was not statically resolved.

2) Entry class interconnection, i.e., RPC transitions to other PPEP (see Section 6.1), usually accumulates all permissions required by the additionally called entry classes for the UID that called the first entry class in the control-flow. However, those transitions are irrelevant for permission analysis when the RPC is located between calls to `Binder.clearCallingIdentity` and `Binder.restoreCallingIdentity`. Clearing the calling UID in the framework’s bound services resets it from the calling app’s UID to the privileged system server UID. Thus, outgoing IPC edges after clearing and before restoring the UID should be ignored in permission analysis, since the additional PPEP are called with a UID that is different from the calling app’s UID.

3) We add a light-weight SDK analysis to reason about required permissions of documented APIs. To this end, we conduct a reachability analysis from public SDK methods to framework EPs (SDK to framework layer in Figure 1). Combining this mapping with the mapping from framework EPs to permissions creates a permission map for the documented API.

7.1 Re-Visiting Permission Mapping

The *Stowaway* project [32] were the first to generate a comprehensive permission map for Android 2.2. Their dynamic analysis approach (feedback directed API fuzzing) generates precise but incomplete results. Moreover, the involved manual effort makes it difficult to re-use it for newer API versions. *PScout* [7] improved on this situation by statically analyzing the framework code, thus increasing the code coverage. In direct comparison *PScout*’s results contain notably more permission mappings. To handle the complexity induced by the framework size, *PScout* resorts to low-precision data models based on class hierarchy information. In the following, we demon-

strate that this has negative implications for their resulting permission map. Using our insights we provide permission mappings that call the validity of prior mappings into question.

We compare our results with *PScout* using their latest available results (for Android 4.1.1)³. Since we exclude `Intent` and `ContentProvider` permissions, which both require supplemental analysis effort such as manifest or `URI` object parsing, we restrict the comparison to un-/documented APIs. For the evaluation we include the standard system apps and make identical assumptions as *PScout*, i.e., we assume that any permission found for a particular API is indeed required (a more precise analysis would require path-sensitivity). Moreover, like *PScout*, we did not conduct a native code analysis.

7.1.1 Documented API map

Figure 3 shows for our documented API map (SDK EP to permissions) how often a certain permission is required. For some permissions *PScout* reports higher numbers while for others AXPLORER reports higher numbers. Since the results are fairly deviating, we manually inspected various cases, including a full analysis of NFC and bluetooth, to verify correctness of our generated numbers. *PScout*’s higher method count, particularly for the two cases of NFC and bluetooth, originates from adding package-protected methods that are not exposed to app developers and from improper handling of the `@hide` javadoc⁴ attribute, resulting in an overcounting of the documented API methods. Our higher numbers of `BROADCAST_STICKY` and `SET_WALLPAPER` mainly refer to abstract methods from the `Context` class that are implemented in its subclass `ContextWrapper` and then inherited by 18 non-abstract subclasses (for API 16). Instead, *PScout* only lists those methods for the `Context/-Wrapper` class, thus missing to count the non-abstract subclasses.

Figure 4 provides a different view of the mapping by showing the distribution of required permissions per API. The main difference is the smaller number of outliers in our data set: four mappings with three or more required permissions, compared to 58 such outliers in the *PScout* data set. While the different results in Figure 3 mainly originate from technical shortcomings in the SDK analysis, Figure 4 hints at the different quality of the undocumented API map as result of a more precise

³We use *PScout*’s results as published on their website at <http://pscouth.cs.toronto.edu>. Last visited 01/25/2016.

⁴EP methods annotated with the `@hide` attribute are not included in the SDK.

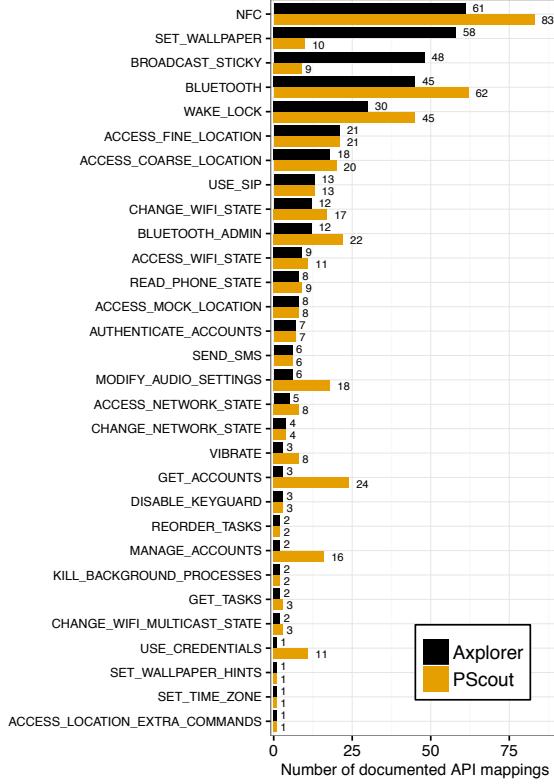


Figure 3: Number of documented APIs per permission.

framework analysis (see next Section 7.1.2). *PScout*'s more light-weight framework analysis results in an over-approximation of permission usage of EPs. For their outliers with more than five permissions in the `ConnectivityManager` class they either overapproximate the receivers of a `sendMessage` call and/or did not resolve the message code and the correct path in the `handleMessage` method. In such cases the over-approximation in the framework analysis negatively influences the quality of the SDK map when IPC calls from the SDK to the application framework are connected. We manually validated all outliers and found that no method actually requires more than three permissions, thus contradicting the *PScout* results. The four outliers in our dataset check at most two permissions, independent of the EP call arguments. Additional permission checks might be required for specific arguments/parameters. For instance the `setNetworkPreference(int)` function of the `ConnectivityService` will tear down a specific type of network trackers depending on a preference integer argument. Some subtypes such as the `BluetoothTetheringDataTracker` require both bluetooth permissions to execute this functionality while other subtypes require no additional permis-

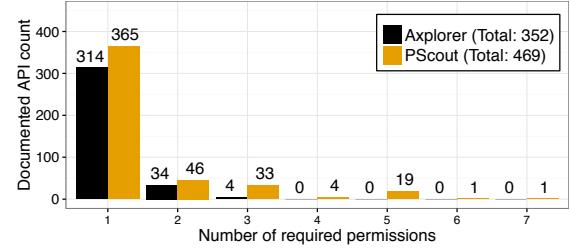


Figure 4: Number of permissions required by a documented API.

sion. Adding parameter-sensitivity to the analysis is required to resolve such cases automatically and to annotate permission checks with conditions.

7.1.2 Undocumented API map

A fair, direct comparison of permission maps for undocumented APIs is unfortunately very difficult due to shortcomings in the original paper. Although *PScout* did not explicitly define the term *undocumented API*, we assume after manual inspection of their results that it refers to the publicly exposed framework interfaces and covers any functionality that can be called from application level (independent of whether it is provided by SDK or system apps). Hence we refer to undocumented API as the entire set of framework entry points (cf. Section 4.1).

In contrast to *PScout*'s documented API map, we discovered different inconsistencies in their undocumented mappings. Besides valid mappings from PPEP to permissions, they also include mappings for unrelated methods. First, public methods of AIDL-based entry classes (which we define as *Entry Points*) are counted up to five times: once in the SDK manager class, in the framework service class, in the AIDL interface class, and in the auto-generated `Stub` and `Proxy` classes. Second, their mapping contains methods of `StateMachine` classes. `StateMachines` are used framework-internally and their functionality is not exposed to apps. Third, synthetic accessor methods as well as methods of anonymous inner classes are reported. We assume that this problem is related to the lack of a concise entry point definition that induces difficulties with the abort criteria during their backwards analysis starting from permission checks. In contrast, our forward analysis seems more suitable in this context, as permission checks are usually closely located to framework EPs.

Table 1 reports on the numbers of entry points per API level. For Android 4.1.1, AXPLORER found 863 PPEP (33.4% of entry points) that require at least one permission. These numbers include signature/

OrSystem permissions since this information, although not interesting for app developers, is of interest for understanding the Android permission model in its entirety. On average we found 1.17 permissions per PPEP, which leads to a total of 1,012 permission mappings that cover 129 distinct permissions. This is a magnitude less than the 32,304 permission mappings reported by *PScout* for normal and dangerous permissions only. However, due to our more concise definition of what constitutes public framework functionality and the inclusion of all permission levels, we argue that our number is more substantiated.

7.2 Permission Locality

The application framework implements a separation of duty: every bound service is responsible for managing a certain system resource and enforcing permissions on access by apps to them. For instance, the `LocationService` manages and protects location related information or the `PhoneInterfaceManager` facilitates and guards access to the radio interfaces. Permission strings already convey a meaning of the kind of system resource they protect and app developers might have an intuition where those permissions are required. We study whether permission checking also follows the principle of separation of duty and permissions are checked by only one particular service. We call this aspect *permission locality*. A low permission locality indicates that a certain permission is enforced at different (possibly unrelated) services. This potentially contributes to the app developer’s permission *incomprehension* that can lead to overprivileged apps [32]. Moreover, a strict separation of duty, i.e., high permission locality, significantly eases the task of implementing (and verifying) authorization hooks for resources, for instance in the design of recent security APIs [21, 8]. Consequently, the permission that protects a set of sensitive operations is ideally checked only in one associated entry class.

To study the permission locality, we analyze the checked permission strings and map them to the enclosing class of the permission check call. In Android v4.1.1 (API level 16) we found that out of 110 analyzed permissions 22 (20%) are checked in more than one class. Among these permissions, 13 are checked in two classes, 5 in three classes and 4 in four classes. An example for seemingly unrelated classes are `LocationManagerService` and `PhoneInterfaceManager` that both check the dangerous permission `ACCESS_FINE_LOCATION`. While the permission is intuitively related to the first service, the connection to the latter one becomes only obvious by looking at the enclosing method that

includes the check (e.g. `getCellLocation`). Interestingly, `PhoneInterfaceManager` is not a framework service but included in the telephony system *app*. Mixing framework services and system apps for enforcing identical permissions complicates permission validation and policy enforcement, since system apps might be vendor-specific. Grouping permissions by protection level results in 22.2% (12/54) of normal/dangerous permissions and 17.9% (10/56) of signature/-OrSystem permissions being checked in distinct classes. This implies that low permission locality equally affects all protection levels. Applying this analysis on API 22 results in an even lower overall permission locality. Focusing on the four outliers in API 16, changes in API 22 include three class renamings, two removals and nine additions (cf. Figure 6 in Appendix C). The permission `CONNECTIVITY_INTERNAL` more than doubled the number of classes (10) in which it is enforced. This evolution of permission checks indicates a disconcerting trend to lower permission locality.

Instead, the permission locality should be increased by, ideally, associating each permission with a single service. Once a designated owner service has been identified for each permission, a dedicated permission check function could be publicly exposed via its `Binder` interface, e.g., a method to check the `ACESSS_FINE_LOCATION` permission could be added to the `ILocationManager` interface. The addition and removal of callers to such methods then no longer affects the number of decision points and preserves the separation of duty for permission checks.

8 Discussion of Other Use-Cases

We briefly discuss further use-cases that can benefit from our work, particularly from our taxonomy of protected resources and the insights from our permission locality analysis.

Permission check inconsistencies. Prior work *Kratos* has shown that the default permission check is inconsistent and can lead to attacks [33]. However, this approach explicitly did not make the attempt to identify protected resources in Android’s application framework but instead relied on arbitrary shared code as heuristic to identify security relevant hotspots in the framework’s code base. While this approach has successfully demonstrated the need for such analysis, we argue that using our definition of protected resources as refinement of shared code can further improve the precision of their analysis, since, by definition, protected resources describe sensitive operations. False positives originating from shared logging

or library code are automatically eliminated then. Distilling a more concise definition of field-update and native method call resources from our high-level taxonomy is a promising future work. An example for such refinement is the removal of non-relevant field updates of a `this` reference within constructors. As there is no prior state for this object, such updates must not be flagged as protected resource.

Authorization hook placement. Different Android framework extensions [26, 46, 29, 12, 21, 8] augment the application framework with authorization hooks in a best effort approach. On commodity systems, a comparable situation for the Linux and BSD kernels has been improved through a long process that established a deeper understanding of the internal control and data flows of those kernels and that allowed development of tools to verify or automate placement of authorization hooks. A similar evolution for Android’s application framework has yet been precluded due to open technical challenges: first, one must be able to analyze control and data flows in the framework across process and service boundaries; second one must be able to track the execution state of the framework service along its internal control and data flows (e.g., tracking the availability of the subject identity); third, one has to establish a clear and very specific understanding of the protected resources of each service. This work at hand addresses the first of these challenges and provides necessary permission locality information to implement comprehensive, coarse-grained enforcement models. Additionally, with our high-level taxonomy of protected resources we made a first step towards solving the third challenge.

9 Conclusion

In this paper, we studied the internals of the Android application framework, in particular challenges and solutions for static analysis of the framework, and provided a first high-level classification of its protected resources. We applied our gained insights to improve on prior results of Android permission mappings, which are a valuable input to different Android security research branches, and to introduce permission locality as a new aspect of the permission specification. Our results showed that Android permission checks violate the principle of separation of duty, which might motivate a more consolidated design for permission checking in the future. To allow app developers and independent research to benefit from our results, we published our data sets as

well as lint rules for our permission mappings for the Android Studio IDE at <http://www.axplorer.org>.

Acknowledgments

This work was supported by the German Federal Ministry for Education and Research (BMBF) under project VFIT (16KIS0345) and SmartPriv (16KIS0377K) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and the initiative for excellence of the German federal government.

References

- [1] Soot - Java Analysis Framework. <http://sable.github.io/soot/>, 1999.
- [2] T.J. Watson Libraries for Analysis (WALA). <http://wala.sf.net>, 2006.
- [3] ANDERSON, J. P. Computer security technology planning study, volume ii. Tech. Rep. ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Oct. 1972.
- [4] ANDROID DEVELOPER DOCUMENTATION. Bound services. <http://developer.android.com/guide/components/bound-services.html>. Last visited: 05/08/2015.
- [5] ARZT, S., BODDEN, E., AND RASTHOFER, S. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proc. 21th Annual Network and Distributed System Security Symposium (NDSS ’14)* (2014), The Internet Society.
- [6] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND McDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation (PLDI 2014)* (2014).
- [7] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: Analyzing the android permission specification. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS ’12)* (2012), ACM.
- [8] BACKES, M., BUGIEL, S., GERLING, S., AND VON STYPEREKOWSKY, P. Android Security Framework: Extensible multi-layered access control on Android. In *Proc. 30th Annual Computer Security Applications Conference (ACSAC ’14)* (2014), ACM.
- [9] BACKES, M., BUGIEL, S., HAMMER, C., SCHRANZ, O., AND VON STYPEREKOWSKY, P. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proc. 24th USENIX Security Symposium (SEC ’15)* (2015), USENIX.
- [10] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., AND VON STYPEREKOWSKY, P. Appguard - enforcing user requirements on Android apps. In *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’13)* (2013).
- [11] BLACKSHEAR, S., GENDREAU, A., AND CHANG, B.-Y. E. Droidel: A general approach to android framework modeling. In *Proc. ACM SIGPLAN Workshop on State of the Art in Program Analysis (SOAP’15)* (2015), ACM.

- [12] BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proc. 22nd USENIX Security Symposium (SEC '13)* (2013), USENIX Association.
- [13] CAO, Y., FRATANTONIO, Y., BIANCHI, A., EGELE, M., KRUEGEL, C., VIGNA, G., AND CHEN, Y. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS '15)* (2015), ISOC.
- [14] CHAUDHURI, A., FUCHS, A., AND FOSTER, J. SCanDroid: Automated security certification of Android applications. Tech. Rep. CS-TR-4991, University of Maryland, 2009.
- [15] EDWARDS, A., JAEGER, T., AND ZHANG, X. Runtime verification of authorization hook placement for the Linux security modules framework. In *Proc. 9th ACM Conference on Computer and Communication Security (CCS '02)* (2002), ACM.
- [16] GANAPATHY, V., JAEGER, T., AND JHA, S. Automatic placement of authorization hooks in the Linux Security Modules framework. In *Proc. 12th ACM Conference on Computer and Communication Security (CCS '05)* (2005), ACM.
- [17] GIBLER, C., CRUSSELL, J., ERICKSON, J., AND CHEN, H. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. 5th international conference on Trust and Trustworthy Computing (TRUST '12)* (2012), Springer-Verlag.
- [18] GORDON, M. I., KIM, D., PERKINS, J. H., GILHAM, L., NGUYEN, N., AND RINARD, M. C. Information flow analysis of android applications in DroidSafe. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS '15)* (2015), ISOC.
- [19] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *Proc. 36th International Conference on Software Engineering (ICSE '14)* (2014), pp. 1025–1035.
- [20] GRACE, M., ZHOU, W., JIANG, X., AND SADEGHI, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC '12)* (2012), ACM.
- [21] HEUSER, S., NADKARNI, A., ENCK, W., AND SADEGHI, A.-R. Asm: A programmable interface for extending android security. In *Proc. 23rd USENIX Security Symposium (SEC '14)* (2014), USENIX.
- [22] HUANG, H., ZHU, S., CHEN, K., AND LIU, P. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *Proc. 22nd ACM Conference on Computer and Communication Security (CCS'15)* (2015), ACM.
- [23] JEON, J., MICINSKI, K. K., VAUGHAN, J. A., FOGL, A., REDDY, N., FOSTER, J. S., AND MILLSTEIN, T. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In *Proc. 2nd ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '12)* (2012), ACM.
- [24] LI, L., BARTEL, A., BISSYANDÉ, T. F., KLEIN, J., LE TRAON, Y., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND McDANIEL, P. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proc. 37th International Conference on Software Engineering (ICSE '15)* (2015).
- [25] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS '12)* (2012), ACM.
- [26] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *Proc. 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS '10)* (2010), ACM.
- [27] OCTEAU, D., LUCHAUP, D., DERING, M., JHA, S., AND McDANIEL, P. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proc. 37th International Conference on Software Engineering (ICSE '15)* (2015).
- [28] OCTEAU, D., McDANIEL, P., JHA, S., BARTEL, A., BODDEN, E., KLEIN, J., AND LE TRAON, Y. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *Proc. 22Nd USENIX Conference on Security (SEC '13)* (2013), USENIX Association.
- [29] ONGTANG, M., MC LAUGHLIN, S. E., ENCK, W., AND McDANIEL, P. Semantically rich application-centric security in Android. In *Proc. 25th Annual Computer Security Applications Conference (ACSAC '09)* (2009), ACM.
- [30] PANDITA, R., XIAO, X., YANG, W., ENCK, W., AND XIE, T. Whyper: Towards automating risk assessment of mobile applications. In *Proc. 22nd USENIX Security Symposium (SEC '13)* (2013), USENIX.
- [31] PEARCE, P., PORTER FELT, A., NUNEZ, G., AND WAGNER, D. AdDroid: Privilege separation for applications and advertisers in Android. In *Proc. 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)* (2012), ACM.
- [32] PORTER FELT, A., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proc. 18th ACM Conference on Computer and Communication Security (CCS '11)* (2011), ACM.
- [33] SHAO, Y., OTT, J., CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS '16)* (2016), ISOC.
- [34] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. Ad-split: Separating smartphone advertising from applications. In *Proc. 21st USENIX Security Symposium (SEC '12)* (2012), USENIX Association.
- [35] SONG, D., ZHAO, J., BURKE, M. G., SBIRLEA, D., WALLACH, D., AND SARKAR, V. Finding tizen security bugs through whole-system static analysis. *CoRR abs/1504.05967* (2015).
- [36] TAN, L., ZHANG, X., MA, X., XIONG, W., AND ZHOU, Y. Autoises: Automatically inferring security specifications and detecting violations. In *Proc. 17th USENIX Security Symposium (SEC '08)* (2008), USENIX.
- [37] THE ANDROID OPEN-SOURCE PROJECT. Security-Enhanced Linux in Android. <http://source.android.com/devices/tech/security/selinux/index.html>. Last visited: 07/27/2015.
- [38] VIDAS, T., CHRISTIN, N., AND CRANOR, L. F. Curbing android permission creep. In *Proc. Workshop on Web 2.0 Security and Privacy 2011 (W2SP 2011)* (2011).

- [39] WATSON, R., MORRISON, W., VANCE, C., AND FELDMAN, B. The TrustedBSD MAC Framework: Extensible kernel access control for FreeBSD 5.0. In *Proc. FREENIX Track: 2003 USENIX Annual Technical Conference* (2003), USENIX Association.
- [40] WEI, F., ROY, S., OU, X., AND ROBBY. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. 21th ACM Conference on Computer and Communication Security (CCS '14)* (2014), ACM.
- [41] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux Security Modules: General security support for the Linux kernel. In *Proc. 11th USENIX Security Symposium (SEC '02)* (2002), USENIX Association.
- [42] WU, L., GRACE, M., ZHOU, Y., WU, C., AND JIANG, X. The impact of vendor customizations on android security. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13)* (2013), ACM.
- [43] YANG, Z., AND YANG, M. Leakminer: Detect information leakage on Android with static taint analysis. In *Proc. 2012 Third World Congress on Software Engineering (WCSE '12)* (2012), IEEE Computer Society.
- [44] ZHANG, X., EDWARDS, A., AND JAEGER, T. Using equal for static analysis of authorization hook placement. In *Proc. 11th USENIX Security Symposium (SEC' 02)* (2002), USENIX.
- [45] ZHANG, Y., YANG, M., XU, B., YANG, Z., GU, G., NING, P., WANG, X. S., AND ZANG, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13)* (2013), ACM.
- [46] ZHOU, Y., ZHANG, X., JIANG, X., AND FREEH, V. Tampering information-stealing smartphone applications (on Android). In *Proc. 4th International Conference on Trust and Trustworthy Computing (TRUST '11)* (2011), Springer-Verlag.

Appendix

A Entry Class Interconnection

A standard call graph gives information about the enclosing method/class of function calls. However, this information is insufficient to provide information about the interconnection of framework entry classes. Instead, the interesting information is the originating entry class that leads to an RPC rather than the actual class that encloses the RPC. To provide better information on the RPC dependencies of entry classes, AXPLORER creates an RPC map by recording RPCs to other entry classes and mapping them to the original entries during control-flow slicing. Figure 5 shows a subgraph of the overall RPC interconnections between flows from different entry classes on Android 5.1 that AXPLORER generated. Nodes correspond to entry classes and are weighted by in-/out-degree, thus highlighting highly-dependent classes such as `ActivityManagerService`. The source of a directed edge is the originating entry class: the control-flow starts at an entry method of this class and at some point along the flow, not necessarily in the same class, an RPC to the class of the edge target node is invoked.

Across all four investigated Android versions there is a median number of three distinct RPC receivers per entry class in our map. The `ActivityManagerService` is an exceptional case where flows from its entry methods reach 36 different entry classes. These numbers emphasize that large parts of the framework are strongly connected and that detailed knowledge about the communication behavior greatly simplifies further framework analyses as explained above.

B Evaluation of protected resources

B.1 Statistics on protected resources

Table 2 provides absolute numbers of protected resources by API version and the distribution by resource types.

B.2 Manual investigation of RTE

Due to the surprisingly high number of runtime exceptions and the fact that uncaught RTE might potentially crash the system, we manually investigated the corresponding code locations to better understand their security implications. The

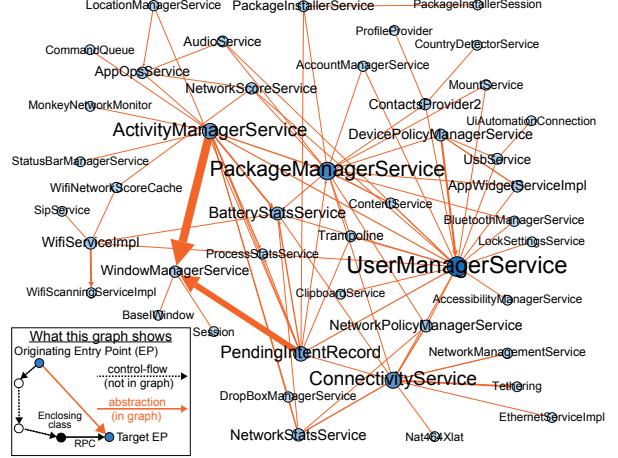


Figure 5: Subgraph of overall entry class interconnection via RPCs in Android 5.1. Directed edges show an RPC to an exposed `IInterface` method of target node.

source code analysis revealed that these instructions reside both in Android’s code base and in external library code. Android, similar to other operating systems, relies on external libraries for specific tasks, such as XMLPullParser, Bouncy Castle, and J-SIP. Integrating library code into such a complex system usually raises the question on how to handle unchecked exceptions that are thrown by library functions. Either the library code is patched, which might be a tedious maintenance task, or exceptions are caught at caller site. Although the latter case is considered bad practice in case of runtime exceptions, using generic catch handler around library call sites is the easiest and most reliable approach in this situation.

Runtime exceptions in the Android code are thrown to indicate precondition violations, like crucial class fields being null, unrecoverable IO errors, or security violations. The implications of a runtime exception differ with respect to code location in which they are thrown. Exceptions in the application layer, i.e. in system applications, cause the respective app to crash. Similarly as for normal apps, they can be restarted by the user. Sticky system app services are restarted automatically by the system. Exceptions thrown in bound services of the `SystemServer` are handled in a special way: A system watchdog thread `com.android.server.Watchdog` constantly monitors the core services like `PowerManagerService` and `ActivityManagerService`. In case of a crash or deadlock, it reboots the entire system. Uncaught runtime exceptions in unmonitored system services cause the Android Runtime, including Zygote and

Android version	4.1.1 (16)	4.2.2 (17)	4.4.4 (19)	5.1 (22)
# of protected resources	6,490	6,969	7,488	10,044
- field updates	4,891 (75.36%)	5,268 (75.59%)	5,675 (75.79%)	7,520 (74.87%)
- native method calls	1,433 (22.08%)	1,499 (21.51%)	1,643 (21.94%)	2,305 (22.95%)
- return value	51 (0.79%)	60 (0.86%)	54 (0.72%)	69 (0.69%)
- broadcast sender	65 (1.00%)	72 (1.03%)	53 (0.71%)	78 (0.78%)
- throw runtime exception	50 (0.77%)	70 (1.01%)	63 (0.84%)	72 (0.71%)
Median context depth	8	9	11	9

Table 2: Numbers on protected resources by type and Android version.

the SystemServer to be restarted (hot reboot) while the kernel keeps running—effects also described in recent research [22].

C Statistics on permission locality

Table 3 shows a detailed statistic about the number of permissions in API 16 that are checked in more than one class, grouped by their protection level. These numbers imply that there is no apparent correlation about the relative frequency and the protection level, in particular between the permissions available to third-party apps (normal+dangerous) and the ones reserved for the system. Hence, low permission locality cases occur across all permission protection levels.

Protection level	# permissions
normal	2/16 (12.5%)
dangerous	10/38 (26.3%)
signature	2/25 (8%)
signatureOrSystem	8/31 (25.8%)

Table 3: Number of permissions in API 16 that are checked in more than one class grouped by permission protection level.

Figure 6 lists all four permissions that are checked in four distinct entry classes in API level 16 (colors denote changes in API 22). In case of READ_PHONE_STATE, those four classes even reside in four distinct packages of which one is part of the telephony system app. Although it may not always be easy to identify one dedicated service as the permission owner, an effort might be desirable to centralize permission checks into as few services as possible (in best case into a single service). Besides renamed classes (blue color) as result of a code refactoring process, the number of additions and removal for this small number of examples clearly confirms that permission checks are violating the separation of duty and underline the need for central enforcement points.

```

Permission : ACCESS_NETWORK_STATE
Level      : normal
Checked in :
- com.android.server.ConnectivityService
- com.android.server.ethernet.EthernetServiceImpl
- com.android.server.ThrottleService
- com.android.server.net.NetworkPolicyManagerService
- com.android.server.net.NetworkStatsService

Permission : READ_PHONE_STATE
Level      : dangerous
Checked in :
- com.android.internal.telephony.PhoneSubInfoProxy
- com.android.phone.PhoneInterfaceManager
- com.android.internal.telephony.SubscriptionController
- com.android.server.TelephonyRegistry
- com.android.server.net.NetworkPolicyManagerService

Permission : CONNECTIVITY_INTERNAL
Level      : signatureOrSystem
Checked in :
- com.android.server.ConnectivityService
- com.android.server.NetworkManagementService
- com.android.server.NsdService
- com.android.server.net.NetworkPolicyManagerService
- com.android.server.net.NetworkStatsService
- com.android.server.ethernet.EthernetServiceImpl
- com.android.server.connectivity.Tethering
- com.android.bluetooth.pan.PanService$BluetoothPanBinder
- com.android.server.wifi.WifiServiceImpl
- com.android.server.wifi.p2p.WifiP2pServiceImpl

Permission : UPDATE_DEVICE_STATS
Level      : signatureOrSystem
Checked in :
- com.android.server.power.PowerManagerService$BinderService
- com.android.server.LocationManagerService
- com.android.server.am.BatteryStatsService
- com.android.server.wifi.WifiServiceImpl
- com.android.server.am.UsageStatsService

```

Figure 6: Permissions checked in four distinct classes in API 16. Colors denote changes in API 22: renamed classes (blue), additions (green) and removals (red).

Practical DIFC Enforcement on Android

Adwait Nadkarni, Benjamin Andow, William Enck
{anadkarni,beandow,whenck}@ncsu.edu
North Carolina State University

Somesh Jha
jha@cs.wisc.edu
University of Wisconsin-Madison

Abstract

Smartphone users often use private and enterprise data with untrusted third party applications. The fundamental lack of secrecy guarantees in smartphone OSes, such as Android, exposes this data to the risk of unauthorized exfiltration. A natural solution is the integration of secrecy guarantees into the OS. In this paper, we describe the challenges for decentralized information flow control (DIFC) enforcement on Android. We propose context-sensitive DIFC enforcement via *lazy polyinstantiation* and practical and secure network export through *domain declassification*. Our DIFC system, *Weir*, is backwards compatible by design, and incurs less than 4 ms overhead for component startup. With *Weir*, we demonstrate practical and secure DIFC enforcement on Android.

1 Introduction

Application-based modern operating systems, such as Android, thrive on their rich application ecosystems. Applications integrate with each other to perform complex user tasks, providing a seamless user experience. To work together, applications share user data with one another. Such sharing exposes the user’s private and enterprise information to the risk of exfiltration from the device. For example, an email attachment opened in a third party document editor (e.g., *WPS Office*) could be exported if the editor was malicious or compromised.

Android’s permission framework is used to protect application data. However, permissions are only enforced at the first point of access. Data once copied into the memory of an untrusted application can be exported. This problem is generic in OSes that provide only data protection, but not data secrecy, and can be solved by integrating information flow secrecy guarantees.

Classic information flow control (IFC) [8] only captures well-known data objects through a centralized policy. On Android, data is often application-specific (e.g.,

email attachments, notes). Therefore, Android requires decentralized IFC (DIFC) [21, 26, 44, 49], which allows data owners (i.e., applications) to specify the policy for their own data objects.

Although DIFC systems have been proposed for Android [19, 28, 46], existing enforcement semantics cannot achieve both security and practicality. For instance, an Android application’s components are instantiated in the same process by default, even when executing separate user tasks. Since the various secrecy contexts from the tasks share state in process memory, DIFC enforcement on the process is hard, as the combined restrictions from all secrecy contexts would make individual components unusable. Prior approaches solve this problem by eliminating Android’s default behavior of application multitasking, and in ways detrimental to backwards compatibility, i.e., 1) killing processes per new call, which could result in dangling state, or 2) blocking until the application voluntarily exits, which could lead to deadlocks.

Similarly, different secrecy contexts may share state on storage through common application files (e.g., application settings). Proposals to separate this shared state on storage (e.g., Maxoid [46]) either deny access to application resources or require applications to be modified. To summarize, prior DIFC proposals for Android cannot separate shared state in memory and on storage while maintaining security and backwards compatibility.

In this paper, we present *Weir*,¹ a practical DIFC system for Android. *Weir* allows data owner applications to set secrecy policies and control the export of their data to the network. Apart from the data owners, and applications that want to explicitly use *Weir* to change their labels, all other applications can execute unmodified. *Weir* solves the problem of shared state by separating memory and storage for different secrecy contexts through *polyinstantiation*. That is, *Weir* creates and manages instances of the application, its components, and its stor-

¹Weir: A small dam that alters the flow of a river.

age for each secrecy context that the application is called from, providing availability along with context-sensitive separation. Our model is transparent to applications; i.e., applications that do not use *Weir* may execute oblivious to *Weir*'s enforcement of secrecy contexts.

We term our approach as “*lazy*” *polyinstantiation*, as *Weir* creates a new instance of a resource only if needed, i.e., if there is no existing instance whose secrecy context matches the caller's. Additionally, *Weir* provides the novel primitive of *domain declassification* for practical and secure declassification in Android's network-driven environment. Our approach allows data owners to articulate trust in the receiver of data (i.e., trusted network domain). This paper makes the following contributions:

- We identify the challenges of integrating DIFC into Android. Using these challenges, we then derive the goals for designing DIFC enforcement for Android.
- We introduce the mechanism of “*lazy*” *polyinstantiation* for context-sensitive separation of the shared state. Further, we provide the primitive of *Domain Declassification* for practical declassification in Android's network-driven environment.
- We design and implement *Weir* on Android. *Weir* incurs less than 4ms overhead for starting components. *Weir*'s design ensures backwards compatibility. We demonstrate *Weir*'s utility with a case study using the *K-9 Mail* application.

While *Weir* presents a mechanism that is independent of the actual policy syntax, our implementation uses the policy syntax of the Flume DIFC model [21]. *Weir* extends Flume by allowing implicit label propagation, i.e., *floating labels*, for backwards compatibility with unmodified applications. Since floating labels are *by themselves* susceptible to high bandwidth information leaks [8], we show how *Weir*'s use of floating labels is inherently resistant to such leaks. Note that while language-level IFC models [40–42] often incorporate checks that prevent implicit flows due to floating labels, our solution addresses the challenges faced by OS-level floating label DIFC systems [19, 44]. Finally, we note that *Weir* provides practical DIFC enforcement semantics for Android, and the usability aspect of DIFC policy and enforcement will be explored in future work.

In the remainder of this paper, we motivate the problem (Section 2), and describe the challenges in integrating DIFC on Android (Section 3). We then describe the design (Section 4), implementation (Section 5) and security (Section 6) of *Weir*, followed by the evaluation (Section 7), and a case study (Section 8). We then discuss the limitations (Section 9), related work (Section 10) and conclude (Section 11).

2 Motivation and Background

We now motivate the need for data secrecy on Android. This is followed by background on DIFC and Android.

2.1 Motivating Example and Threat Model

Consider Alice, an enterprise user in a BYOD (bring your own device) context. Alice receives an email in the enterprise *OfficeEmail* application with an attached report. She edits the report in a document editor, *WPS Office*, and saves a copy on the SD card, accessible to all applications that have the READ_EXTERNAL_STORAGE permission. Later, Alice uses the *ES File Explorer* to browse for the report, edits it in *WPS Office*, and then shares it with *OfficeEmail* to reply to the initial email.

To perform their functions, untrusted third party data managers such as *ES File Explorer* require broad storage access. Even without direct access, user-initiated data sharing grants data editors like *WPS Office* access to confidential data. If *ES File Explorer* or *WPS Office* were malicious or compromised, they could export Alice's confidential data to an adversary's remote server.

Threat Model and Assumptions: We seek to enable legitimate use of third party applications to process secret user data, while preventing accidental and malicious data disclosure to the network. For this purpose, our solution, *Weir*, must mediate network access, and track flows of secret data 1) among applications and 2) to/from storage.

Weir's trusted computing base (TCB) consists of the Android OS (i.e., kernel and system services), and core network services (e.g., DNS). *Weir* assumes a non-rooted device, as an adversary with superuser privileges may compromise OS integrity. Further, we assume correct policy specification by the data owner applications, specifically regarding declassification. To prevent timing and covert channels based on shared hardware resources (e.g., a hardware cache), the only alternative is denying data access to secret data or the shared resource. *Weir* does not defend against such channels, which are notoriously hard to prevent in DIFC OSes in general.

2.2 Why Information Flow Control (IFC)?

Android uses its permission framework to protect user data. While permissions provide protection at the first point of access, the user or the data owner application (e.g., *OfficeEmail*) have no control over the flow of data once it is shared with another application (e.g., *WPS Office*). Unauthorized disclosure is an information flow problem that permissions are not designed to solve.

Information flow control (IFC) [8] can provide data secrecy and prevent unauthorized disclosure, through the definition and enforcement of the allowable data flows in the system. In an IFC system, subjects (e.g., processes)

and objects (e.g., files) are labeled with predefined security classes (e.g., top-secret, secret, confidential). The secrecy policy determines the data flow (i.e., ordering) between any two classes based on a partially ordered finite lattice. Labels may also be joined to form a higher label in the lattice. For secrecy, data can only flow up, i.e., to a higher security class [6], and violating flows require declassification by the policy administrator.

2.3 What is DIFC?

A centralized IFC policy can only describe the secrecy constraints for well-known data objects (e.g., location, IMEI). Decentralized IFC (DIFC) [26] extends the IFC lattice to include unknown subjects and objects, and is appropriate for protecting application-specific data, such as Alice’s secret report received by *OfficeEmail*. We now describe some fundamental aspects of DIFC.

Label Definition: In a DIFC system, security principals create labels (i.e., security classes) for their own secret data. On Android, decentralized label definition would allow apps to control the flow of their data by creating and managing labels for their data. Note that while DIFC also provides integrity, our description is for data secrecy as it is the most relevant to the problem in Section 2.1.

Label Changes and Floating Labels: The finality of subject and object label assignment is called *tranquility* [6], a property of mandatory protection systems. Tranquility constraints have to be relaxed for DIFC policy. Subjects may then change (raise or lower) their labels “safely”, i.e., with authorization from the data owners whose security classes are involved in the change.

Explicit label changes offer flexibility over immutable labels, but are not practical in environments where communication is user-directed and unpredictable a priori. *Floating* labels (e.g., in Asbestos [44]) make DIFC compatible with unmodified apps in such cases, by allowing seamless data flows through implicit label propagation. That is, the caller’s and the callee’s labels are joined, and the resultant label is set as the callee’s label.

Declassification: The network is considered to be public, and any network export requires declassification by the data owner. Data owners may choose to explicitly declassify every request to export their data, or allow trusted third parties to declassify on their behalf. While the former is impractical when frequent declassification is required, the latter bloats the data owner’s TCB.

System Integration: One of the first steps while integrating data secrecy into an existing OS is the selection of the subject for data flow tracking. Fine-grained dynamic taint tracking (e.g., TaintDroid [13]) labels programming language objects to provide precision, but does not protect against implicit flows. OS-based DIFC

approaches [21, 49] adopt the better mediated OS process granularity, but incur high false positives; i.e., functions sharing the process with unrelated functions that read secret data may be over-restricted. While secure process-level labeling is desired, practical DIFC enforcement must minimize its impact on functionality.

2.4 Android Background

The Android application model consists of four components, namely *activities* for the user interface (UI), *services* for background processing, *content providers* to provide a uniform interface to application data, and *broadcast receivers* to handle broadcast events.

Component Instantiation: Services and content providers run in the background, and have one active instance. Activities can have multiple instances, and the default “standard” launch behavior for activities is to start a new instance per call. Developers use Android’s “`android:launchMode`” manifest attribute to manage activity instances as follows: *SingleTop* activities are resumed for new calls if they already exist at the top of the activity stack. *SingleTask* and *SingleInstance* activities are similar in that they are allocated an instance in a separate user task and every call to such an activity resumes the same instance; the only difference being that the latter can be the only activity in its task.

Inter-Component Communication: Inter-component communication on Android can be 1) indirect or 2) direct. Indirect communication is an asynchronous call from one component to another, through the *Activity Manager* service (e.g., `startActivity`, `bindService`). Direct communication involves a synchronous *Binder* remote procedure call (RPC) to the callee using the callee’s “*Binder object*”. While direct communication bypasses the Activity Manager, its setup involves one mediated indirect call to retrieve the callee’s Binder object. For example, the first operation executed on a content provider (e.g., `query`, `update`) by a caller is routed through the Activity Manager, which retrieves the content provider’s Binder object and loads it into the caller’s memory. Future calls are routed directly to the content provider.

3 DIFC Challenges on Android

In this section, we discuss the four aspects of Android that make DIFC enforcement challenging. Further, we describe how previous Android DIFC systems fare with respect to the challenges, and state the design goals for practical DIFC enforcement on Android.

1. Multitasking on Android: Android’s UI is organized into user tasks representing the user’s high-level objectives. An application can be involved in multiple tasks

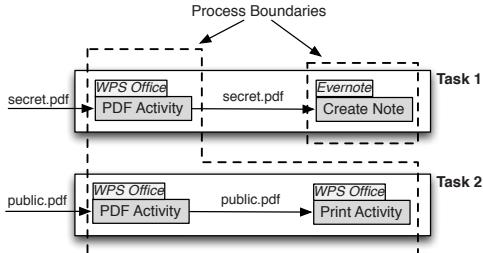


Figure 1: Shared state in memory: Instances of *WPS Office* performing different tasks in the same process.

by default. Further, a default activity can be instantiated multiple times in one or more concurrent tasks [3].

Figure 1 shows two tasks. In *Task 1*, the user opens a secret PDF (e.g., a contract) with *WPS Office*, which loads it in its *PDF Activity*, and shares it with the *Evernote* app. In *Task 2*, the user opens a non-confidential PDF (e.g., a published paper) in another instance of *WPS Office*'s *PDF Activity*. Further, in *Task 2*, the user chooses to print the PDF, which is then sent to *WPS Office*'s internal *Print Activity*. As seen in Figure 1, multiple activities of the *WPS Office* app as well as multiple instances of the *PDF Activity* run in the same process, and may share data in memory (e.g., via global variables).

As the two instances of the *PDF Activity* are instantiated with data of different secrecy requirements (i.e., *secret.pdf* and *public.pdf*), they run in different *secrecy contexts*. Enforcing the DIFC policy on the process due to the sensitive nature of *Task 1* would also unjustifiably restrict the non-sensitive *Task 2*. A naive solution of forcing every component to start in a separate process may break components; e.g., *Print Activity* may try to access a global variable initialized by the *PDF Activity*, and may crash if the *PDF Activity* is not in the same process. To summarize, component instances in various secrecy contexts often share state in process memory, making process-level enforcement challenging.

2. Background components: As described in Section 2.4, service and content provider components have only one active instance, which is shared among all of an application's instances, and may also communicate with other applications. As a result, various secrecy contexts may mix in a single background component instance.

If floating labels (described in Section 2.3) are applied, then the background component may accumulate the labels of all the secrecy contexts it communicates with, and then propagate its new label back to the components connected to it. This results in a *label explosion*, where the entire system acquires a large, restrictive label that cannot be declassified by any single security principal. Note that background components may run in the shared application process by default. Therefore, restarting a background component's process for each new call is infeasible, as it would crash the other components (e.g., a

foreground activity) running in that process.

3. Internal and External Application storage: Android provides each application with its own internal (i.e., private) storage shared amongst all of its runtime instances, irrespective of the secrecy context. For example, both the sensitive and non-sensitive instances of *WPS Office* may access the same user settings in the application's private directory. For availability from all secrecy contexts, storage access enforcement uses floating labels. The propagation of sensitive secrecy labels through shared application files (i.e., shared state on storage) may cause label explosion. The risk and impact of label explosion is higher on the external storage (i.e., the SD card) shared by all applications.

4. Internet-driven environment: Android applications are often connected to the Internet. In such an environment with frequent network export, explicit declassification by the data owner is inefficient. Delegation of the declassification privilege to allow export without the owner's intervention would bloat the application's TCB. Additionally, existing declassification mechanisms described in Section 2.3 make the policy decision based on the identity of the security principal performing the export. On Android, such mechanisms would limit the user to using a small subset of applications for data export (i.e., out of the 2 million applications on Google Play [39]), which would be detrimental to adoption of DIFC on Android.

3.1 Prior DIFC Proposals for Android

We discuss three prior DIFC proposals for Android, namely Aquifer [28], Jia et al. [19] and Maxoid [46], all of which are OS-level DIFC systems. Our objective is to understand the design choices made by these systems, with respect to the challenges described previously.

1. Aquifer: Our prior work, Aquifer [28], provides protection against accidental data disclosure, by tracking the flow of data through applications, and enforcing the declassification policy for network export.

For seamless data sharing between applications, Aquifer uses the floating labels described in Section 2.3. To limit label explosion, Aquifer does not label background components, and hence can only prevent accidental data disclosure. On the other hand, Aquifer labels storage, but does not claim to mitigate label explosion on storage. Further, to prevent different secrecy requirements for data in the memory of a single process, Aquifer disables Android's multi-tasking and restarts the process of the existing instance when the application is called from another secrecy context. Finally, Aquifer's declassification policy allows the data owner to explicitly specify the security principal that may export data,

or a condition on the call chain for implicit export.

2. Jia et al.: The DIFC system by Jia et al. [19] also uses floating labels to support general-purpose applications, but supports strict secrecy policies (i.e., relative to Aquifer) that may restrict data sharing among applications if needed.

Contrary to Aquifer, the system propagates labels to background components, providing stronger protection against malicious data exfiltration. At the same time, the system makes no claims of controlling label explosion via background components or storage. The system uses Flume’s capabilities [21] for declassification. This work also acknowledges the challenge of multi-tasking along with DIFC enforcement, and disallows multi-tasking by blocking new calls to an application until all of its components voluntarily exit. Since Android components do not exit by themselves like conventional OS programs, such blocking could potentially lead to deadlocks.

3. Maxoid: Xu and Witchel [46] provide an alternate approach to file system labeling to prevent label explosion in Maxoid, by using file system polyinstantiation [22] to separate differently labeled data on disk.

Maxiod addresses new calls to existing labeled instances in a manner similar to Aquifer’s; i.e., by restarting the instance. Additionally, Maxoid prevents access to background components from labeled instances, thereby preventing label explosion, although at the cost of backwards compatibility. On the other hand, Maxoid considers overt data flows through Binder IPC as declassification, unlike the system by Jia et al. that mediates such communication. Finally, Maxoid modifies system content providers (e.g., Contacts) to use a SQL proxy, in order to extend its label separation into system content providers. As a result, Maxoid’s storage separation is unavailable for use by content providers in unmodified third party applications.

Takeaways: Prior approaches demonstrate the possibility of DIFC on Android, and make convincing arguments in favor of using floating labels, mainly for backwards compatibility with Android’s unpredictable data flows. At the same time, we observe that in prior systems it becomes necessary to relax either security or backwards compatibility in order to use floating labels on Android (e.g., with background components). Additionally, prior approaches recognize the need to separate different secrecy contexts in process memory, but the proposed solutions disable Android’s default multi-tasking.² Finally, in systems that aim to address label explosion on storage, only separating the shared state on storage without addressing the shared state in memory may be insufficient to support unmodified applications.

²Killing existing processes or blocking can result in the killing of unrelated components sharing the process, or deadlocks, respectively.

3.2 Design Goals

Our objective is to design DIFC enforcement that provides security, and is backwards compatible with unmodified applications. Our design goals are as follows:

- G1** *Separation of shared state in memory.* DIFC enforcement must ensure that data from different secrecy contexts is always separated in memory, preferably in the memory of different processes. Process-level enforcement can then be used to mediate flows between differently labeled data.
- G2** *Separation of shared state on storage.* DIFC enforcement must ensure that data from different secrecy contexts is separate on persistent storage. For mediation by the OS, the separation must be at the level of OS objects (e.g., files, blocks).
- G3** *Transparency.* A naive implementation of goals **G1** and **G2** would affect the availability of components and storage. Our system must be transparent, i.e., applications that do not use the DIFC system must be able to operate oblivious to the enforcement.
- G4** *Secure and practical declassification for network export.* A DIFC system on Android should provide a declassification primitive that is both feasible (i.e., does not hinder the use of applications) and secure.

4 Weir

In this paper, we propose *Weir*, a practical and secure DIFC system for Android. *Weir*’s design is guided by the security and backwards compatibility goals described in Section 3.2. We now briefly describe the specific properties expected from our design, followed by an overview of *Weir* and design details.

Design Properties: Taking a lesson from prior work in Section 3.1, our system must allow seamless data sharing between applications for backwards compatibility with Android’s application model. Data flows must be tracked using implicit label propagation (i.e., floating labels), while mitigating the risk of label explosion. More specifically, our system must not deny data access, unless an application explicitly changes its label and fails a label check. Since our goal is to prevent unauthorized data export, network access may be denied if an application tries to export sensitive data to the network in violation of the declassification requirements of the data owner. Finally, our system must mediate all overt data flows, but covert channels existing in Android are not the targets of our system (discussed further in Section 9).

4.1 Overview

In *Weir*, applications define the policy for their data by creating their own *security classes*. *Weir* labels files (as

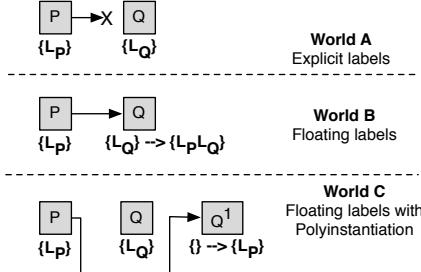


Figure 2: Overview of floating labels w/ polyinstantiation relative to explicit and floating labels.

objects) and processes (as subjects), granting the kernel complete mediation over all data flows among subjects and objects. As *Weir*'s contributions are in its policy-agnostic mechanism, we use the generic terminology from Section 2.3 in policy-related discussions. Section 5.1 describes our implementation's policy model.

Weir uses floating labels (described in Section 2.3), as explicit labels are hard to assign a priori in Android, where data flows are often user-directed and unpredictable. However, naive (i.e., context-insensitive) floating label propagation can cause certain components to acquire more labels due to involvement in multiple secrecy contexts, and eventually become unusable. We propose polyinstantiation to make floating labels context sensitive, and hence separate the shared state from different secrecy contexts in memory (**G1**) and on storage (**G2**). Our approach is in principal similar to context sensitive inter-procedural analysis that adds precision by considering the calling context when analyzing the target of a function call (e.g., summary functions and call strings [36], k-CFA [37], and CFL-reachability [31]). To our knowledge, context sensitivity has not been explored in the scenario investigated in this paper. Further, the approach of secure multi-execution [11] also uses multiple executions of the program, but is fundamentally different in many aspects, as we describe in Section 10.

We describe polyinstantiation relative to explicit and floating labels with the example scenario in Figure 2, where an instance of component *P* with label $\{L_P\}$ tries to send a message to an instance of component *Q* with a label $\{L_Q\}$, and where $\{L_P\} \neq \{L_Q\}$. In World A where only explicit labels are allowed, the message would be denied as *Q* would not be able to explicitly change its label to $\{L_P\}$ without a priori knowledge of *P*'s intention to send a message. In World B with floating labels, the flow would be automatically allowed, with *Q*'s new label implicitly set to a join of the two labels. While World B allows seamless communication, it does not prevent the two secrecy contexts (i.e., $\{L_P\}$ and $\{L_Q\}$) from mixing, leading to the challenges we explored in Section 3. In World C, we use polyinstantiation along with floating labels, and a new instance of *Q* denoted as Q^1 is created

in the caller's context (i.e. with the caller's label $\{L_P\}$), separate from the original instance of *Q* with label $\{L_Q\}$. Thus, our approach allows the call to take place, without the mixing of secrecy contexts. The “lazy” aspect of our approach (not represented in the figure) is that we would reuse a previously created instance of *Q*, denoted Q^{past} , if its label matched the caller's label (i.e., $\{L_P\}$). Additionally, while the new instance has an empty label (i.e., $\{\}$) as the base (compile-time) label in our prototype, our model can be adapted to support a different base label.

Weir uses lazy polyinstantiation for all indirect inter-component calls (e.g., starting an activity, querying a content provider) (described in Section 2.4). *Weir* polyinstantiates processes, Android components and the file system, creating new instances of each for different secrecy contexts. Floating labels allow legacy apps to integrate into *Weir* without modification for making or receiving calls, while polyinstantiation adds context sensitivity. *Weir*'s use of floating labels supports process-level labeling along with application multi-tasking (**G3**), a more practical solution than the alternatives of killing existing instances [28, 46] or indefinite blocking [19].

We now describe *Weir*'s polyinstantiation of memory, followed by storage. We then discuss how *Weir* supports explicit label changes. Finally, we describe how *Weir*'s domain declassification satisfies goal **G4**.

4.2 Polyinstantiation of Memory

To satisfy goal **G1**, *Weir* must ensure that no two component instances with mismatching labels execute in the same process. At the same time, *Weir* must make components available if the underlying Android enforcement (i.e., permission framework) allows. Therefore, our approach polyinstantiates both components and processes to make them available in multiple secrecy contexts.

For backwards compatibility, our approach refrains from affecting developer configurations (e.g., by forcing the “multi-process” manifest attribute). Instead, *Weir* polyinstantiates components within the application’s own context. Specifically, *Weir* upholds the process assignments made for components by the developer, through the “android:process” manifest attribute (i.e., the component’s *processName*). That is, *Weir* ensures that components that were meant to run together (i.e., assigned the same *processName*), still run together. We now describe our approach, followed by an example.

Our approach: On every call, *Weir* retrieves the label of the caller (i.e., the *callerLabel*). *Weir* then checks if an instance of the desired component is running in a process whose label matches *callerLabel*. If one is found, the call is delivered to the matching instance. If not, *Weir* creates a new instance of the called component.

When the target component instance is assigned, *Weir*

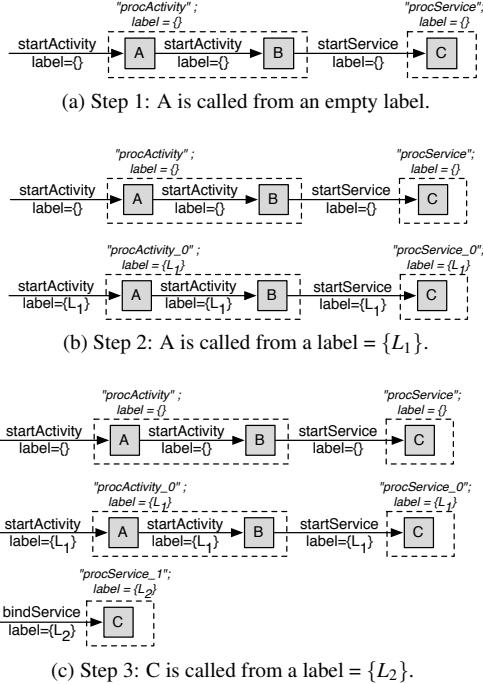


Figure 3: *Weir*'s lazy polyinstantiation of three app components; activities A and B, and a service C.

must find a process to execute it. If the process associated with this component (i.e., *processName*) has a different label, *Weir* cannot execute the new instance in it, and has two options: a) assign a polyinstantiated process that is associated with *processName* and has the label *callerLabel* or 2) create a new process associated with the *processName* with label *callerLabel*. As it is evident based on the first option, *Weir* keeps track of a *processName* and all its instances created for various secrecy labels. *Weir* can then reuse a previously instantiated process that is associated with the original *processName* and already has the required label (i.e., *lazy* polyinstantiation). Additionally, *Weir* can ensure adherence to the developer's process assignment; i.e., that component instances only execute in the process associated with their *processName*. If a matching process is not available, *Weir* creates a new process for *callerLabel*, and internally maintains its association with the original *processName*.

Example: Consider an app with three components, activities A and B, and a service C. The developer sets the *processName* for A and B to be “*procActivity*”, whereas the *processName* for C is set to “*procService*”. This means that A and B are expected to run in the same process, while C runs in a separate process. The app is programmed such that when A is started, it starts B, following which B starts C. Using Figure 3, we describe *Weir*'s instantiation of A, B, and C and their processes.

In Step 1 (Figure 3a), A is first called by an unlabeled caller; i.e., the *callerLabel* is empty. A new instance of A

is created, and a new process by the name “*procActivity*” is started for it. Then, A calls B. The label of A's process is empty, so B is also instantiated with an empty label, in the matching process, i.e., “*procActivity*”. B then calls C, which is instantiated in the new process “*procService*”.

In the Step 2 (Figure 3b), A is called from a caller with *callerLabel* = {*L*₁}. *Weir* cannot deliver the call to the existing instance of A, as its process has a mismatching label (i.e., *callerLabel* = {*L*₁} ≠ {}). Thus, *Weir* creates a new instance of A for this call. As there are no processes associated with “*procActivity*” and with the label {*L*₁}, *Weir* also allocates a new process “*procActivity_0*” to host this instance. Thus, for this call, a new instance of A is started in a new process “*procActivity_0*”, whose label is set to {*L*₁}. When this instance of A calls B, the call is treated as a call to B with *callerLabel* = {*L*₁}, the caller being A's new instance with label {*L*₁}. As *Weir* keeps records of all the processes created for polyinstantiation, it starts a new instance of B in the process that is associated with B's original process “*procActivity*”, and has a matching label {*L*₁}, i.e., “*procActivity_0*”. Reusing an existing process instance is an example of “*lazy*” polyinstantiation. When this instance of B starts C, *Weir* creates a new instance of C due to mismatching labels, in a new process “*procService_0*” with label {*L*₁}.

In Step 3 (Figure 3c), *bindService* is called on C with the label *callerLabel* = {*L*₂}. Since the caller's label {*L*₂} mismatches with the two existing instances of C that are running with labels {} and {*L*₁}, a new instance of C is created. As there are no processes associated with “*procService*” that have a label matching {*L*₂}, a new process “*procService_1*” is created to host the new instance. Note that all of these instances and processes exist simultaneously, as shown in the figures. If C is called again with the label *callerLabel* = {*L*₂}, *Weir* will not have to create a new instance, and the call will be delivered to the existing instance of C running in process “*procService_1*” with the matching label {*L*₂} (i.e., “*lazy*” polyinstantiation).

Weir's approach maintains context-based separation in memory (**G1**), and also ensures that components configured to run in the same process still run together; i.e., our approach is transparent to the application, satisfying goal **G3**. For example, instances of A and B exist together, both in the labeled as well as the unlabeled contexts. *Weir* supports all Android components declared in the application manifest, i.e., activities, services, content providers and broadcast receivers. An exception is broadcast receivers registered at runtime, which are instantiated at registration in the secrecy context of the registering process, and hence not subject to further instantiation. Any future broadcasts to such receivers are treated as direct calls subject to strict DIFC label checks.

4.3 Polyinstantiation of Storage

To prevent restrictive labeling of shared storage by processes running in sensitive contexts, *Weir* extends context-based separation to the storage as well (**G2**). *Weir* achieves this separation without denying access to instances in sensitive secrecy contexts (**G3**).

Our approach: *Weir* separates shared state in the internal and external storage using file-system polyinstantiation via a *layered* file system approach [29]. Our approach is similar to Solaris Containers [22], and more recently, Docker [24]. Context-sensitive storage separation has also been used previously in DIFC, for *known* persistent data objects. For example, in their DIFC system for the Chromium Web browser, Bauer et al. create context-specific copies of bookmarks to prevent a restrictive label from making bookmarks unusable [5].

In *Weir*, every secrecy context receives its own copy-on-write file system layer. Processes running in a particular secrecy context have the same view of the file system, which may be different from those running in other contexts. All file operations are performed on the context-specific layer attached to a process, which relays them to the underlying file system (i.e., the default layer). Unlabeled processes are assigned the default layer.

For simplicity, new layers are always created from the default layer, and never from existing labeled layers. That is, for any layer with label L , the copy-on-write always occurs from the default layer (with label $\{\}$), and not another lower layer (say label L_1), even if L_1 is lower than L (i.e., $L_1 \subseteq L$). An alternate design choice of using a non-default lower layer for copy-on-write could lead to conflicts due to incompatible copies of data at different, but similarly labeled lower layers. For example, two labels L_2 and L_3 might be at the same level below L in the DIFC lattice, but may have different copies of the same file. For resolving such conflicts, the system may have to either involve the user or the application. The backwards compatibility and usability effects of choosing a lower layer need further exploration, although it may be a more flexible option. Hence, our design simplifies the potential choice between contending layers by always choosing to copy from the default layer.

For efficiency, a layer only stores the changes made to the default layer by processes in the layer’s secrecy context (i.e., copy-on-write). When a file present in the default layer is first written by a process attached to a non-default layer, the file is first copied to the non-default layer and then modified. Future accesses for the file from that context are directed to its own copy. When a process attached to a non-default layer tries to read a file that has never been modified in the calling process’s layer, it reads the original file on the default layer. *Weir*’s storage approach is an extension of its *lazy* polyinstantiation, i.e.,

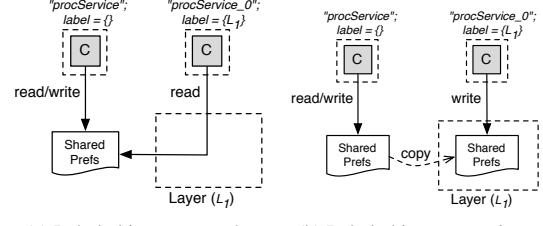


Figure 4: *Weir*’s storage polyinstantiation using layers.

new layers are created only when a process with a previously unknown secrecy context is initialized. Applications transparently access storage using any file system API, and *Weir* directs the accesses to the correct files.

Weir stores the files copied to layers in layer-specific copy-on-write directories. While creating such directories, *Weir* accounts for the security and availability requirements of applications and users. For application-specific internal storage, copy-on-write directories are created in an area that is accessible only to instances of the particular application. For public external storage, *Weir* creates common label-specific copy-on-write directories in an area accessible to all apps. This approach ensures that when an application is uninstalled, its data on external storage is still available to the user.

Example: Figure 4 shows two instances of the component C , one of which is running in the unlabeled secrecy context (i.e., label $\{\}$), while the other has a label $\{L_1\}$. *Weir* sets up a file system layer, i.e., $Layer(L_1)$, to mediate all file accesses by the labeled instance. $Layer(L_1)$ is only attached to processes with label $\{L_1\}$.

As seen in Figure 4a, initially, both the unlabeled and labeled instances of C read from the shared preferences file (i.e., *SharedPrefs*). That is, $Layer(L_1)$ relays all the read requests by the labeled instance of C for unmodified *SharedPrefs* file to the default storage. Once the labeled instance of C attempts to write to the *SharedPrefs* file, *Weir* copies it to $Layer(L_1)$. This copy is used for all future read or write accesses by instances with label $\{L_1\}$.

Security of copy-on-write directories: *Weir* ensures the security of the layered directories through a combination of file labeling and Linux permissions. File labels are initialized when first written, to the writing process’s label. *Weir* uses strict DIFC label checks for all successive file accesses. Further, *Weir* prevents the implicit flows due to the presence or number (i.e., count) of such copy-on-write directories. To address flows through the presence of specific copy-on-write directories, *Weir* uses random directory names known only to the system. To prevent flows that make use of the number of such directories, *Weir* creates the copy-on-write directories inside a parent directory owned by *Weir*. The Linux permissions of this parent directory are set to deny the read operation on it, and hence cannot be used to list or count subdirectories.

Together with the polyinstantiation in memory, *Weir*'s approach enables transparent separation of different secrecy contexts without modifying legacy apps.

4.4 Label Changes and Binder checks

A component instance's label is implicitly set when it is instantiated. Similarly, a file's label is initialized when it is first written to. For all successive accesses (i.e., direct Binder IPC and file reads/writes respectively), *Weir* does not apply floating labels, but performs a strict DIFC label check (i.e., data may not flow to a "lower" label). Hence, any label changes after initialization can only be explicit.

An application aware of *Weir* may change the label of its instances by raising it (to read secret data), or lowering it (to declassify data), provided the change is legal with respect to the policy for the security classes involved in the change, as described in Section 2.3. For example, to read secret data labeled with label $\{L_1\}$, a component instance may raise its label to $\{L_1\}$, if it has authorization (e.g., a capability) from the owner of $\{L_1\}$ (see Section 5.1 for the policy syntax). We now describe the problem caused by explicit label changes.

Problem of explicit label change: A component instance may establish Binder connections with other instances through the Activity Manager, and then use direct Binder RPC. When an instance changes its label, its existing Binder RPC connections (established via indirect communication, see Section 2.4) may be affected. That is, its new label may be higher or lower relative to the instances it is connected to. Hence, it may not be able to send or receive data on existing connections due to the strict DIFC check on Binder transactions. An explicit label change may also make the component instance's context inconsistent with its attached storage layer. At the same time, explicit label changes are unavoidable in applications that use *Weir*.

Our solution: *Weir* provides applications with the *intent labeling* mechanism, i.e., components can label calls (i.e., intent messages), before they are sent to the Activity Manager service, ensuring that the target component is instantiated with the label set on the intent. In fact, a component may instantiate itself with the desired secrecy label by specifying itself as the intent's target. Intent labeling eliminates the need for explicit label changes.

Security of Intent Labeling: *Weir* does not blindly trust the label set on the intent, as applications may otherwise abuse the mechanism for unauthorized declassification. For example, a malicious component with the label $\{L_1\}$ may add secret data to an intent, and set an empty label (i.e., $\{\}$) on the intent before calling itself with it. To account for such malicious use cases, *Weir* checks if the calling application would be authorized to explicitly change its current label to the label on the intent, as per

the policy (see Section 5.1). A call with a labeled intent may proceed only if the caller passes the check.

While we have not encountered use cases that cannot be expressed using intent labeling, our implementation allows explicit label changes, mainly for expert developers who may want to make temporary changes to their instance labels. Explicit label changes must be used with caution, as our design does not account for the problems due to label change after instantiation (e.g., dropped Binder calls), since labels do not float to existing Binder connections and files to prevent label explosion (explained in Section 3).

4.5 Domain Declassification

Problems with traditional network declassification are rooted in the decision to declare trust in the exporting subject, as discussed in Section 3. More precisely, in an internet-driven environment, it may be more practical for the data secrecy enforcement to reason about *where* the data is being delivered, rather than *who* is performing the export. *Weir* introduces the alternative of *domain declassification* to allow data owners to articulate trust in terms of the receiver, i.e., the target Web domain. *Weir* allows the data owner to associate a set of network domains (t^D) with its security class (t). When the data in context $\{t\}$ is to be exported to the network, *Weir*'s enforcement implicitly declassifies t , if the destination domain is in t^D . The data owner is neither required to explicitly declassify nor trust the exporting application.

In Section 8, we discuss an example where the enterprise only wants data to be exported to a set of enterprise domains, irrespective of the application exporting it. Such a policy allows the user to use the same email application for both the personal and work account, but prevents accidental export of work data to the personal SMTP server. Domain declassification not only addresses the goal of practical declassification in a network driven environment (**G4**), but also prevents the user from accidentally exporting data from a trusted application, but to an untrusted server.

Weir is not the first IFC system to use domains for declassification, although most prior systems to do so consider domains as security principals (e.g., COWL [41], Bauer et al. [5]). For instance, COWL confines JavaScript using a declassification policy analogous to the well-known same origin policy (SOP), i.e., code executes in the context of its origin, and hence possesses the declassification privilege for export to the origin's Web domain. In this case, the origin Web domain is a first class security principal, as it has physical presence on the device in the form of the code running in its context. Thus, in COWL, the declassification privilege is still expressed in terms of the security principal that is sending the data (i.e., the origin). On Android, there is

no direct correlation between Web domains and applications; i.e., Web domains do not have code executing in their context on the device, and hence are not security principals. Thus, *Weir*'s approach of expressing trust in the receiver of the data (i.e., the Web domain) rather than the sender is indeed unique among OS-level DIFC systems where Web domains may not be security principals [1, 21, 28, 49]. Hails [15], an IFC web framework for user privacy, may be closer to *Weir*'s approach, as it allows users to declassify their data for specific domains. Hails users are prompted to explicitly declassify when network requests to disallowed domains are first made, which may not be feasible on Android (see Section 3).

Weir's enforcement is limited to the device, and may not defeat an adversary controlling the network. While we leave this aspect relaxed for our threat model, we note that DNSSEC or IPsec could be used in such scenarios.

5 Implementation

We implemented *Weir* on Android v5.0.1, and the Android Kernel v3.4. This section describes the essential aspects of our implementation. The source code can be found at <http://wspr.csc.ncsu.edu/weir/>.

5.1 Weir's DIFC Policy

Weir derives its policy structure from the Flume DIFC model [21], which consists of *tags* and *labels*. A *data owner* (O) application defines a security class for its sensitive data in the form of a secrecy tag (t). A set of tags forms a secrecy label (S). *Weir* enforces the IFC secrecy guarantee, i.e., “no read up, no write down” [6]. Information can flow from one label to another only if the latter dominates, i.e., is a superset of the former. For instance, data can flow from a process P to a process Q if and only if $S_P \subseteq S_Q$. *Weir* applies this strict DIFC check to direct Binder communication and file accesses.

Each tag t has associated capabilities, namely t^+ (for reading) and t^- (for declassification), which data owners delegate to specific apps, or all other apps (i.e., the global capability set G). At any point of time, a process P has an effective capability set composed of the capabilities delegated to its application (C_P), and the capabilities in G . P can change its label S_P to S_P^+ by adding a tag t if and only if $t^+ \in C_P \cup G$. Similarly, P can change its label S_P to S_P^- by removing a tag t if and only if $t^- \in C_P \cup G$.

As the network interface is untrusted, it has an empty label, i.e. $S_N = \{\}$. Thus, a process P must have an empty label (i.e., $S_P = \{\}$), or the ability to change its label to $S_P = \{\}$ to create a network connection, i.e., $\forall t \in S_P, t^- \in C_P \sqcup G$. Additionally, *Weir* extends Flume's syntax with the domain declassification capability t^D , which is a set of trusted Web domains for tag t specified by the owner O . For a network export to a domain $d \in t^D$, t is implicitly declassified.

5.2 Component Polyinstantiation

When a component calls (i.e., sends an intent message or queries a content provider), the Activity Manager resolves the target component to be called using the static information present in the application manifest. *Weir* does not interfere with this *intent resolution* process. Then, the Activity Manager chooses the actual runtime instance of the resolved component, which is where *Weir*'s polyinstantiation takes effect. That is, *Weir* controls component instantiation, without modifying the components themselves. Hence, *Weir* is compatible with all developer manifest options, except ones that control instantiation. Section 7.2 provides a compatibility evaluation for such options. For a detailed explanation on Android's component startup workflow and *Weir*'s component instantiation logic, see Appendix A.

5.3 File-system Layering

We chose OverlayFS [29] over alternatives (e.g., aufs), as it is in the Linux kernel (since v3.18). As the current OverlayFS patch is incompatible with SELinux, we set SELinux to monitoring mode. This is a temporary limitation, as OverlayFS developers are working towards full integration [45], which is on SELinux's Kernel ToDo list as well [10]. Additionally, we could use a fine-grained block-level copy-on-write file system (e.g., BTRFS [32]). There are advantages to using such file systems, as we describe in the trade-offs (Section 9). Note that while we could get the Android Linux kernel to compile with BTRFS, the build system support tools that are required to build Android's sparse-images for BTRFS (e.g., `ext4_utils` for ext4) are missing. Therefore, our prototype opts for OverlayFS, as it does not require user-space support.

5.4 Process Initialization

On Android, the *zygote* process forks and prepares new processes for applications. When a new process is forked, *Weir* sets its secrecy label in the kernel, and uses zygote to mount the appropriate storage *layer* to the process's mount namespace based on its label. If the process has a non-empty label, *Weir* separates the process's mount namespace from the global mount namespace using the *unshare* system call, and mounts the appropriate OverlayFS copy-on-write layer based on the label on top of the unlabeled file system. New layers are allocated when new labels are first encountered. *Weir* maintains the mapping between a label, its assigned layer and the specific copy-on-write directories used for it.

5.5 Kernel Enforcement

Weir uses a Linux security module (LSM) to track the security contexts of processes and files in the kernel. We

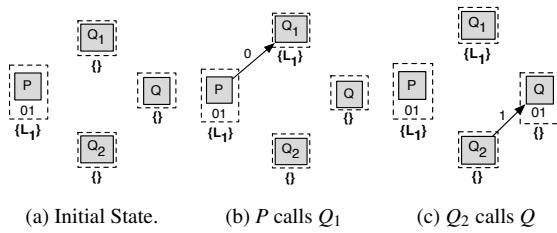


Figure 5: Floating label DIFC system: Q receives 1 and guesses 0 for every reply not received.

integrated the multi-LSM patch [35] to enable concurrent SELinux and *Weir* enforcement. The security context of a process contains its secrecy label and capabilities, while that of a file only contains a secrecy label. We now describe the enforcement for file access, Binder communication and network access, as follows:

1. Files: *Weir* uses the *file_permission* LSM hook to mediate each file read and write access. The secrecy label of a file (stored in the xattrs) is initialized from the label of the process that first writes it.

2. Binder: *Weir* mediates Binder transactions in the kernel using the Binder LSM hooks. For compatibility, *Weir* whitelists Binder communication with Android system services in the kernel. To prevent apps from misusing whitelisted services as implicit data channels, we manually analyzed all system service API, and modified API that may be misused, e.g., the Clipboard Manager service provides label-specific clipboards in *Weir*.

3. Network: *Weir* mediates the socket *connect* and *bind* operations in the kernel. The tags in the calling process’s label that cannot be declassified using its capability set are sent to *Weir*’s system service in the userspace via a synchronous upcall, along with the IP address of the destination server. *Weir*’s system service then resolves the domain name from the IP address, which is challenging, as a reverse DNS lookup may not always resolve to the same domain used in the initial request. Fortunately, Android proxies all DNS lookups from applications to a separate system daemon. We modify the daemon to notify *Weir* when a process performs a DNS lookup, including the domain name and the IP address returned. During the domain declassification upcall, this mapping is referenced to identify the destination domain. *Weir* allows the connection only if all the tags in the upcall can be declassified for that domain.

6 Security of Polyinstantiation

Floating labels were first predicted to be prone to information leaks by Denning [8]. While language-level floating label IFC models (e.g., COWL [41] and LIO [40,42]) can mitigate such leaks, securely using floating labels is

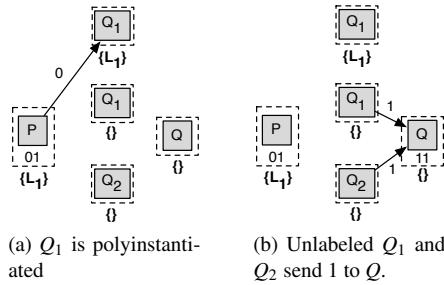


Figure 6: *Weir*: Q always receives data “11”

still a challenge for OS-level DIFC systems (e.g., IX [23] and Asbestos [44]). We discuss an attack on an OS-level floating label DIFC system, described in Krohn and Tromer’s paper on the non-interference of Flume [20], and show how *Weir* is resistant to such data leaks. We use Android’s terminology to describe the attack.

We describe the attack twice; once in a floating label system without polyinstantiation (Figure 5) and once in *Weir* (Figure 6). Figure 5a shows the malicious components (e.g., services) P , Q , Q_1 and Q_2 . P has obtained the data “01”, and the accompanying label $\{L_1\}$. P wants to transfer the data to Q , without Q obtaining the label $\{L_1\}$. Note that Q , Q_1 and Q_2 initially have the empty label $\{\}$. Additionally, P and Q have a prior understanding that P will call the i^{th} service of Q to indicate “0” at the i^{th} bit. Q ’s components are programmed to send a message to Q after a predetermined time if they do not receive a message from P (i.e., indicating a “1”). Since the first data bit is “0”, P sends “0” to Q_1 , whose label floats to $\{L_1\}$ (Figure 5b). After a predefined time, the component that did not receive a message from P , i.e., Q_2 , sends a “1” to Q (Figure 5c). The data leak is successful, as Q knows that the second bit is “1”, and assumes the first to be “0”, all without acquiring the label $\{L_1\}$. As Android does not place any limits on the number of components, a wider n bit channel is possible with n components.

Weir’s polyinstantiation defeats this attack by creating a new instance of Q_1 in a separate process to deliver a call from a label that mismatches its own (Figure 6a). Next, the unlabeled instance of Q_1 and Q_2 both call Q with data “1”, as shown in Figure 6a. In fact, for n components of Q , Q will always get n calls with data “1”, as *Weir* will polyinstantiate all the components that have been called by P with the label $\{L_1\}$. *Weir*’s use of floating labels is resistant to implicit flows inherent to regular floating labels, as labels do not float to the original instance, but to a new instance created in the caller’s context.

Jia et al [19] attempt to solve a similar problem,³ by making the raised label the component’s base (i.e., static) label. This defense allows the existing leak, but makes the components that received the message (e.g., Q_1) un-

³Refer to page 8 of the paper by Jia et al. [19] for details.

usable for future attacks. Attackers can be expected to beat this defense by coordinating the components used for every attack, and transferring significant data before all the components have restrictive static labels.

Finally, while polyinstantiation is resistant to data leaks in floating labels, we leave the complete formalization of this idea as future work. The intuition behind the formalization is described as follows: Let \mathcal{L} be the type system corresponding to the labels (e.g., type-system for floating labels) and \mathcal{S} be the type system corresponding to information about stacks (e.g., for k -CFA analysis strings of size k that capture information about last k calls). Assume that we have inferencing/propagation rules for both type systems and they are sound. We have an intuition that the combined system (denoted by $\mathcal{L} \times \mathcal{S}$) is sound (the inferencing/propagation rules are basically a combination of both rules).

7 Evaluation

Our evaluation answers the following questions about *Weir*'s performance and compatibility:

Q1 Is *Weir* compatible with developer preferences that manipulate component instantiation?

Q2 What is *Weir*'s performance overhead?

Q3 Is *Weir* scalable for starting components?

We now provide an overview of the experiments and highlight the results. The rest of this section describes each experiment in detail.

7.1 Experiment Overview and Highlights

Weir does not modify components, but only modifies their instantiation. Thus, we evaluate compatibility with options that control component instantiation (**Q1**), i.e., the singleTop, singleTask and singleInstance activity launch modes described in Section 2.4. We trigger the launch modes in popular Android apps from Google Play, and record application behavior in unlabeled and labeled contexts. We did not observe any crashes or unexpected behavior. Every launch mode worked as expected, while the underlying polyinstantiation ensured delivery of calls to instances in the caller's context.

We measure the performance overhead of *Weir* over an unmodified Android (AOSP) build (**Q2**) with microbenchmarks for common operations (e.g., starting components). Our comparison between the unmodified build, *Weir* (unlabeled instance), and *Weir* (labeled instance) in Table 1 shows negligible overhead. Even in cases where the overhead percentage is large, the absolute overhead value is negligible (<4ms). Further, the negligible difference in the values of *Weir*'s labeled and unlabeled instances (i.e., relative to the error) would make a noisy covert channel at best.

As described in Section 5, for every call, Android's intent resolution gets the target component. The OS then chooses a runtime instance from available instances of the target. Hence, the total number of a component's runtime instances only affects its own start time. We evaluate the scalability of a component's start time, when a certain number of its instances already exist (**Q3**). Our results in Figure 7 show a linear increase in the start time with increase in the number of concurrent instances, and low absolute values (e.g., about 56 ms for 100 instances).

7.2 Compatibility with Launch Modes

We randomly pick 30 of the top applications on Google Play (i.e., 10 per launch mode, complete list available at <http://wspc.csc.ncsu.edu/weir/>).

Methodology: For each launch mode, we first launch each application from two separate unlabeled components, and navigate to the specific activity we want to test. With this step, we confirm that the application and specifically the singleTask/Top/Instance activity works as expected. Without closing existing instances, we start the same application from a labeled context and repeat the prior steps. We record any unexpected behavior.

Observations: We did not observe any unexpected behavior, and activities started in their assigned tasks. In the case of singleTask and singleInstance activities, two instances of the same activity ran in the designated task instead of one; i.e., one labeled and the other unlabeled. Intent messages were delivered to the activity instance with the caller's label. This behavior is compatible with singleTask and singleInstance activities, and also maintains label-based separation in memory.

7.3 Microbenchmarks

We evaluate the performance of the operations affected by *Weir* (i.e., file/network access, component/process start), on a Nexus 5 device. We perform 50 runs of each experiment, waiting 200 ms between runs. Table 1 shows the mean with 95% confidence intervals. Cases with negative overhead can be attributed to the high error in some operations. Specific experimental details are as follows:

Component and Process start: We measure the component start time as the time from the placement of the call (e.g., `startActivity`) till its delivery. The component is stopped between runs. To measure the process start time, we kill the process between subsequent runs. While the overhead percentages may be high (e.g., for providers), the absolute values are low, and would not be noticeable by a user. Further, the process start time that includes file-system layering in zygote shows minimal overhead.

File access: We perform file read and write operations on

Table 1: Performance - Unmodified Android (AOSP), Weir in unlabeled context, Weir in labeled context.

Operation	AOSP (ms)	Weir (ms)		Overhead (ms)	
		Weir w/o label	Weir w/ label	Weir w/o label	Weir w/ label
Activity start	20.06±4.47	22.22±4.69	20.82±4.87	2.16 (10.77%)	0.76 (3.79%)
Service start	13.94±2.87	14.96±2.85	17.36±4.78	1.02 (7.32%)	3.42 (24.53%)
Broadcast Receiver start	12.92±3.96	11.42±4.44	11.86±3.34	-1.5 (-11.6%)	-1.06 (-8.2%)
Content Provider start	4.54±2.28	7.26±5.32	7.9±4.73	2.72 (59.91%)	3.36 (74.01%)
Process start	127.18±5.62	130.28±5.63	132.98±6.66	3.1 (2.44%)	5.8 (4.56%)
File Read (1MB)	42.38±6.05	43.46±5.44	41.32±5.39	1.08 (2.55%)	-1.06 (-2.5%)
File Write (1MB)	46.8±5.79	47.84±5.42	47.16±5.85	1.04 (2.22%)	0.36 (0.77%)
Network	66.98±3.62	65.68±2.78	69.00±7.04	-1.3 (-1.94%)	2.02 (3.02%)

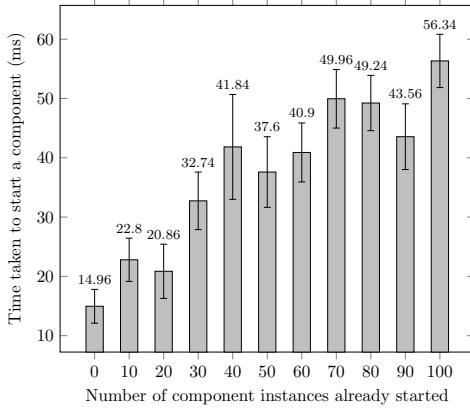


Figure 7: Linear increase in component start time when $0 \rightarrow 100$ instances (in steps of 10) already exist.

a 1MB file using a 8KB buffer. Since the entire check is performed using the process and file labels in the kernel, the overhead value is negligible (e.g., about 0.77% for a labeled file write). We also measure the cost of copying the 1MB file to the labeled layer, i.e., repeating the file write experiment on *Weir* but deleting the file between runs. The extra time taken to copy relative to AOSP is 5.98 ms (about 13% overhead). RedHat’s evaluation of OverlayFS further demonstrates its scalability [18].

Network access: We measure the time to establish a network connection using the *HTTPUrlConnection* API, using domain declassification for the labeled instance. The labeled instance’s overhead includes the kernel upcalls and the DNS proxy lookup. The overhead for the labeled instance (2.02 ms or 3.02%) includes the time taken by the DNS proxy to inform *Weir* of the lookup, as well as the synchronous kernel upcalls.

7.4 Scalability of Component Instantiation

We create up to 100 simultaneous instances (in steps of 10) of a service component, each with a different label. At each step, we then invoke the last instance, i.e., from a caller with the last instance’s label, and measure the component start time. Note that this experiment presents the worst case scenario; i.e., our prototype does not implement any particular strategy (e.g., least recently used (LRU)) for matching a call with a list of available in-

```

1 // Creating the tag 't'
2 domains={ ``www.bcloud.com'', ``smtp.bcloud.com'', ... };
3 createTag(``t'', domains);

```

Listing 1: *BCloud*’s policy configuration

```

1 addTag(``t''); //raise own label to {t}
2 //perform sharing action ...
3 removeTag(``t'')//lower own label

```

Listing 2: *BCloud* raises its label

stances, and a request with the last instance’s label will always result in a label comparison with *all* available instances. Figure 7 shows linear scalability, with the highest absolute value being less than 57 ms.

8 Case Study

We investigated the use of labeled enterprise data with an unmodified third party email (*K-9 Mail*) application [9]. With this case study, we demonstrate *Weir*’s utility, and motivate the trade-off discussion in Section 9.

Application Setup: We created an enterprise cloud application, *BCloud* that allows the user to sync her work data (e.g., contacts, documents) to the device. Further, we used the popular email application *K-9 Mail* with both user and enterprise data. The setup is as follows:

1. *BCloud*. We assume that the enterprise policy is to enable the use of third party applications with work data, but to allow export to only enterprise domains. For example, work data must only be emailed using the work SMTP server *smtp.bcloud.com*.⁴ Thus, *BCloud* creates a tag *t* as shown in Listing 1. To set the policy before sharing its data or saving it to storage, *BCloud* may temporarily raise its label to $\{t\}$ (Listing 2), or start itself or other applications with $\{t\}$ using intent labeling (Listing 3). For instance, *BCloud* raises its label before copying the work contacts to Android’s *Contacts Provider*.

2. *K-9 Mail*. We configured *K-9 Mail* for both personal and work email accounts. Like most modern email clients, *K-9 Mail* allows the user to send an email using the work or the personal account, using the *send as* email field. Internally, *K-9 Mail* uses the SMTP server *smtp.gmail.com* for the personal account, and

⁴We used *mail.yahoo.com*, *smtp.mail.yahoo.com* and *imap.mail.yahoo.com* as *BCloud*’s trusted domains.

`smtp.bcloud.com` for the work account. To assist the user in composing an email, *K-9 Mail* retrieves contacts from the *Contacts Provider* app, and makes suggestions as the user types into the “to” (i.e., sender) field.

Experiment: We opened a document from *BCloud* in the *WPS Office* application. Then, from the *WPS Office* app, we shared the document with *K-9 Mail*. *K-9 Mail*’s “compose” window was displayed. We then chose to *send as* the work account, and picked a contact to add to the “to” field. We tried to attach another file, and the “attach” action opened Android’s system file browser. We selected a file and returned to *K-9 Mail*’s compose screen. We then switched to the home screen without sending the work email. We repeated the entire experiment in the default (i.e., unlabeled) context, with the *send as* field set to the personal account. We then sent both emails. Throughout the experiment, we watched the system log for important events (e.g., network denial).

Observations: We made the following observations, and verified them using the system log:

1. *Context-specific instances.* As we shared work data (in the context $\{t\}$) with *WPS Office* and subsequently *K-9 Email*, instances of these applications (i.e., processes and components) were started in the work context $\{t\}$, and attached to the internal and external (SD card) storage layer $Layer(t)$. The unlabeled context resulted in separate instances with the empty label $\{\}$, attached to the default storage layer. Instances in both contexts existed concurrently, without any crashes or abnormal behavior.
2. *Context-specific data separation.* While attaching another document in the work $\{t\}$ instance of *K-9 Mail*, we could see all the documents on the default storage layer (i.e., unlabeled files), and documents in work $Layer(t)$ (i.e., added from *BCloud*). On the contrary, in the default context, we could only see the files on the default layer. Further, in the default context, *K-9 Mail* suggested from all of the user’s unlabeled contacts, but none of the work contacts. In the work context, *K-9 Mail* suggested from all the work contacts, and the unlabeled contacts that existed before *BCloud* synced its labeled contacts. That is, *K-9 Mail* could not see new records created in the default layer’s contacts database after it was copied over to $Layer(t)$.
3. *Domain Declassification.* In the work context, *K-9 Email* was unable to connect to the SMTP and IMAP sub-domains of `gmail.com`, but could only connect with the domains declassified by tag t . Unmodified *K-9 Email* silently handled these network access exceptions, without crashing or displaying errors messages.

9 Trade-offs and Limitations

This section describes the trade-offs of our approach, motivated in part by the observations in the case study.

```

1 Intent intent = new Intent();
2 // Add 't' to the intent's label.
3 intent.addLabel('t');
4 // Add data to the intent ...
5 startActivity(intent); //Call self

```

Listing 3: *BCloud* starts itself with new label

1. Centralized perspective: The user cannot view both labeled and unlabeled data together, unless an application is started in the labeled context (e.g., *K-9 Mail* in context $\{t\}$). We envision modified application launchers and phone settings that allow the user to start applications (e.g., File Browsers) with a certain label by default, for making labeled and default data available together. Our test apps use similar techniques; hence such launchers should not be hard to create. On the other hand, a centralized perspective on more than one non-default context (e.g., $\{t1, t2, t3, \dots\}$) may require a trusted OS application exempt from polyinstantiation (but subject to only floating labels), as floating labels by themselves are vulnerable to information leaks (Section 6).

2. Updates to default layer: While context-specific versions of files may be generally acceptable, in case of database files (e.g., contacts read by *K-9 Mail* in the work context) the user may expect new records in the unlabeled context to be propagated to the copy in the labeled context. The lack of updates is mainly a trade-off of our file-level copy-on-write implementation (i.e., OverlayFS). As mentioned in Section 5, a block-level copy-on-write file system (e.g., BTRFS [32]) may mitigate this trade-off, as it would only copy the blocks modified by the labeled context, and newly allocated blocks in the default context would be accessible to the labeled context, although this aspect needs further exploration.

3. Access control denials: Floating labels ensure that inter-component communication is never denied, and that resources (e.g., files, other components) are available in all secrecy contexts. Although apps may be denied network access, research has addressed this challenge in the past (e.g., AppFence [17]). Further, most IDEs (e.g., Eclipse) enforce compile-time checks for proper exception handling, and it is uncommon for apps to crash due to network denial, as observed in the case study as well.

4. Instance Explosion: *Weir* creates separate context-specific *K-9 Mail* instances, *only for the contexts in use*. The theoretical worst-case count of component instances is equivalent to the number of components multiplied by the number of all existing contexts (not just those in use). Our event-based and “lazy” instantiation makes this worst case practically improbable, unlike approaches that execute *all existing* contexts (see Section 10). On the other hand, a denial of service attack on a particular application component may be feasible, by starting a very large number of its instances in a short amount of time for noticeable impact on the lookup time of that compo-

ment. Our implementation can be modified to detect and prevent unusual rates of component instantiation. Note that polyinstantiation of a component only affects its own lookup time (as discussed in Section 7.1), and cannot be used for an attack with a device-wide impact.

5. Resource Overhead: Polyinstantiation may cause resource overhead in terms of the memory, battery and storage. The memory overhead is manageable as Android’s out of memory manager automatically reclaims memory from low priority components. Further, any measurement of the battery or storage use is bound to be subjective with respect to the number of labels, number of apps/components, type of apps (e.g., game vs. text editor), aspects of the user scenario (e.g., user-initiated flows, scenario-specific storage access). An objective large-scale study will be explored in the future.

6. Consistency Issues: To a remote server, the instances of an application in *Weir* are analogous to instances running on different devices (e.g., a user logged in from two devices). Hence, any data consistency issues in such scenarios are not a result of polyinstantiation.

7. Covert Channels: *Weir* mediates overt communication between subjects and objects, but does not address covert channels existing in Android. A clearance label [6, 40, 44] can be used to defend against adversaries using covert channels by preventing access to certain tainted data in the first place. While a clearance label can be easily incorporated into *Weir*, setting the clearance policy for third party applications with unpredictable use cases is hard, and needs further exploration from a policy specification standpoint. Finally, unlike IFC systems that focus on preventing untrusted code within a program from exfiltrating data (e.g., Secure multi-execution [11]), *Weir*’s focus is inter-application data sharing. Hence, compartmentalizing an application using clearance is outside the scope of this paper.

8. Explicit labeling of messages and files: On Android, an indirect message through the OS (e.g., intent message) is required before a bi-directional Binder connection can be established between two instances. *Weir* allows floating labels on such indirect communication (but not on direct Binder calls), and polyinstantiation ensures that the two instances at the end of a bidirectional Binder connection have the same label, which is sufficient for synchronous Binder messages. Hence, labeling of individual Binder messages does not provide additional flexibility, unlike in explicit labeling DIFC systems (e.g., COWL [41], Flume [21]). Note that *Weir* allows explicit labeling of indirect messages (i.e., intent labeling). Further, explicit labeling of a file with a label that is different from its creating process instance would place it on an incorrect layer. Such incorrectly stored files will not be visible to future instances started with matching labels,

and may cause unpredictable application behavior. Thus, our design trades the flexibility in explicitly labeling files for stable context-sensitive storage.

10 Related Work

In Section 3.1 we described prior DIFC proposals for Android (i.e., Aquifer [28], Jia et al. [19] and Maxoid [46]). We now describe other relevant prior research.

DIFC: Myers and Liskov presented the Decentralized Labeling Model (DLM) [26] that allowed security principals to define their own labels. Since then, numerous DIFC systems have been proposed that provide valuable policy and enforcement models [20, 21, 25–27, 33, 44, 49, 50]. Language-based DIFC approaches (e.g., JFlow [25] and Jif [27]) provide precision within the program, but rely on the OS for DIFC enforcement on OS objects (e.g., processes, files, sockets). On the contrary, coarse-grained OS-level approaches (e.g., HiStar [49] and Asbestos [44]) provide security for flows between OS objects, but cannot reason about flows at the granularity of a programming language variable. While *Weir* is also an OS-level DIFC approach, which means it cannot achieve precision at the program variable level, context sensitive enforcement ensures that *Weir* always has higher precision than traditional OS-level DIFC. Further, while Laminar [30, 33] provides both language-level as well as the OS-level enforcement, it requires applications to be modified to use the precise language-level enforcement. This is not an option for backwards compatible DIFC on Android. Finally, *Weir* does not require general-purpose applications to explicitly define flows as in Laminar, HiStar and Flume [20, 21], as inter-application communication in Android tends to be unpredictable.

Secure multi-execution: Secure multi-execution [11] was proposed to determine and enforce that a program’s execution is noninterferent, i.e., to eliminate unlawful data flows by untrusted code *within* a program. The approach achieves noninterference using multiple concurrent executions at all points in the lattice, removing statements that do not match the labels of specific executions. On the contrary, lazy polyinstantiation creates only one instance in the security context of the caller. Unlike secure multi-execution where the multiple executions are treated as a part of the same program instance, polyinstantiation treats multiple executions as unrelated context-specific instances separated in memory and storage. Our approach is more suitable for Android’s inter-application data sharing abstractions, while secure multi-execution may be useful to prove non-interference for a general program. Further, secure multi-execution only assumes a finite, predefined label set. This assumption is violated in DIFC systems, where the label set is often

large and not known a priori, and executing all labels at once is impractical.

Faceted Execution: Jeeves [48] and Jacqueline [47] ensure that security principals see different views of data based on their secrecy contexts, using a technique defined as faceted execution. The result of *Weir*'s approach is similar; i.e., each security principal can only see data at its own secrecy context. For faceted execution, the copies of data have to be specified by the programmer a priori, which is acceptable if the security of different users using a single program (e.g., a conference submission site) is to be defined. On the contrary, on Android, *Weir*'s approach of allowing applications to operate unmodified, and creating context-specific copies on the go, is more practical. To elaborate, data in terms of *Weir* is not the value(s) of a programming language variable, but the instances of components in memory and file system layers per label. Finally, just like secure multi-execution, faceted execution is more suitable when the IFC lattice is small (e.g., two labels) or finite, and may not be feasible for DIFC, where tags can be created at runtime.

Coarse-grained Containers: Approaches such as Samsung Knox [34] and Android for Work [2] protect enterprise data by isolating groups of applications into different containers. Containers cannot compensate for the lack of data secrecy guarantees, as they do not address threats within the container, i.e., the accidental export of secret data by a trusted application or the potential compromise of a trusted application. Virtual phones (e.g., Cells [4]) are similarly inadequate for data secrecy.

Transitive Enforcement on Android: Android permissions lack transitive enforcement, and are susceptible to privilege escalation attacks [7, 16]. IPC Inspection [14] enforces transitivity by reducing the caller's effective permissions to those of the least privileged component in the call chain. Quire [12] provides the call chain information to applications being called, to prevent confused deputy attacks. Like floating labels, privilege reduction is additive, and may severely restrict shared components.

Fine-grained Taint Tracking on Android: Taint-Droid [13] detects private data leaks via fine-grained taint tracking on Android, but is vulnerable to implicit flows. CleanOS [43] and Pebbles [38] use fine-grained taint tracking on memory and storage to evict and manage private data respectively. For tracking data in databases, both approaches rely on modification to the database library, which may not be secure as the library executes in the memory of the enforcement subject.

11 Conclusion

Android's component and storage abstractions make secure and practical DIFC enforcement challenging. To ad-

dress these challenges, we present *lazy polyinstantiation* and *domain declassification*. We design and implement a DIFC system, *Weir*, and show a negligible performance impact as well as compatibility with legacy applications. In doing so, we show how secure and backwards compatible DIFC enforcement can be achieved on Android.

Acknowledgements

This work was supported in part by the NSA Science of Security Lablet at North Carolina State University, NSF CAREER grant CNS-1253346, NSF-SaTC grants CNS-1228782 and CNS-1228620, and the United State Air force and Defense Advanced Research Agency (DARPA) under Contract No. FA8650-15-C-7562. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

- [1] ALJURAI DAN, J., FRAGKAKI, E., BAUER, L., JIA, L., FUKUSHIMA, K., KIYOMOTO, S., AND MIYAKE, Y. Run-Time Enforcement of Information Flow Properties on Android. Tech. Rep. CMY-CyLab-12-015, CyLab, Carnegie Mellon University, 2012.
- [2] ANDROID. Android for Work. <https://www.android.com/work/>.
- [3] ANDROID DEVELOPERS . Tasks and Back Stack. <https://developer.android.com/guide/components/tasks-and-back-stack.html>.
- [4] ANDRUS, J., DALL, C., HOF, A. V., LAADAN, O., AND NIEH, J. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 173–187.
- [5] BAUER, L., CAI, S., JIA, L., PASSARO, T., STROUCKEN, M., AND TIAN, Y. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS)* (Feb 2015).
- [6] BELL, D. E., AND LAPADULA, L. J. Secure Computer Systems: Mathematical Foundations. Tech. Rep. MTR-2547, Vol. 1, MITRE Corp., 1973.
- [7] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege Escalation Attacks on Android. In *Proceedings of the 13th Information Security Conference (ISC)* (2010).
- [8] DENNING, D. E. A Lattice Model of Secure Information Flow. *Communications of the ACM* (1976).
- [9] DEVELOPERS, K.-. M. K-9 Mail. <https://github.com/k9mail>, 2015.
- [10] DEVELOPERS, S. SELinux Kernel ToDo. <https://github.com/SELinuxProject/selinux/wiki/Kernel-Todo>, 2015.
- [11] DEVRIESE, D., AND PIESSENS, F. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy* (May 2010).

- [12] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WAL-LACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium* (2011).
- [13] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., McDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).
- [14] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium* (2011).
- [15] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J. C., AND RUSSO, A. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012).
- [16] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the ISCO Network and Distributed System Security Symposium* (2012).
- [17] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).
- [18] JEREMY EDER. Comprehensive Overview of Storage Scalability in Docker. <https://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/>.
- [19] JIA, L., ALJURAI DAN, J., FRAGKAKI, E., BAUER, L., STROUCKEN, M., FUKUSHIMA, K., KIYOMOTO, S., AND MIYAKE, Y. Run-Time Enforcement of Information-Flow Properties on Android (Extended Abstract). In *Proceedings of the European Symposium on Research in Computer Security (ES-ORICS)* (2013).
- [20] KROHN, M., AND TROMER, E. Noninterference for a Practical DIFC-Based Operating System. In *Proceedings of the IEEE Symposium on Security and Privacy* (2009).
- [21] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information Flow Control for Standard OS Abstractions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (2007).
- [22] LAGEMAN, M., AND SOLUTIONS, S. C. Solaris Containers—What They Are and How to Use Them.
- [23] McILROY, M. D., AND REEDS, J. A. Multilevel security in the UNIX tradition. *Software: Practice and Experience* (1992).
- [24] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* (2014).
- [25] MYERS, A. C. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (1999).
- [26] MYERS, A. C., AND LISKOV, B. A Decentralized Model for Information Flow Control. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (1997).
- [27] MYERS, A. C., AND LISKOV, B. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology* (2000).
- [28] NADKARNI, A., AND ENCK, W. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2013).
- [29] NEIL BROWN. Overlay Filesystem. <https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt>.
- [30] PORTER, D. E., BOND, M. D., ROY, I., MCKINLEY, K. S., AND WITCHEL, E. Practical Fine-Grained Information Flow Control Using Laminar. *ACM Trans. Program. Lang. Syst.* (Nov. 2014).
- [31] REPS, T. W. Program Analysis via Graph Reachability. *Information & Software Technology* 40, 11-12 (1998).
- [32] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)* (Aug. 2013).
- [33] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (2009).
- [34] SAMSUNG ELECTRONICS. An Overview of Samsung Knox. http://www.samsung.com/global/business/business-images/resource/white-paper/2014/02/Samsung_KNOX_whitepaper_June-0-0.pdf, 2013.
- [35] SCHAUFLER, C. LSM: Multiple concurrent LSMs. <https://lkml.org/lkml/2013/7/25/482>, 2013.
- [36] SHARIR, M., AND PNUELI, A. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*. 1981.
- [37] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 1991.
- [38] SPAHN, R., BELL, J., LEE, M., BHAMIDIPATI, S., GEAMBASU, R., AND KAISER, G. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *Proceedings of the USENIX Operating Systems Design and Implementation (OSDI)* (2014).
- [39] STATISTA. Number of available applications in the Google Play Store from December 2009 to February 2016. <http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>
- [40] STEFAN, D., RUSSO, A., MITCHELL, J. C., AND MAZIÈRES, D. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell* (2011), Haskell '11.
- [41] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIÈRES, D. Protecting Users by Confining JavaScript with COWL. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Oct. 2014).
- [42] STEFAN, H., STEFAN, D., YANG, E. Z., RUSSO, A., AND MITCHELL, J. C. IFC Inside: Retrofitting Languages with Dynamic Information Flow Control. In *Proceedings of the 4th Conference on Principles of Security and Trust (POST 2015)* (2015).
- [43] TANG, Y., AMES, P., BHAMIDIPATI, S., BIJLANI, A., GEAMBASU, R., AND SARDA, N. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012).

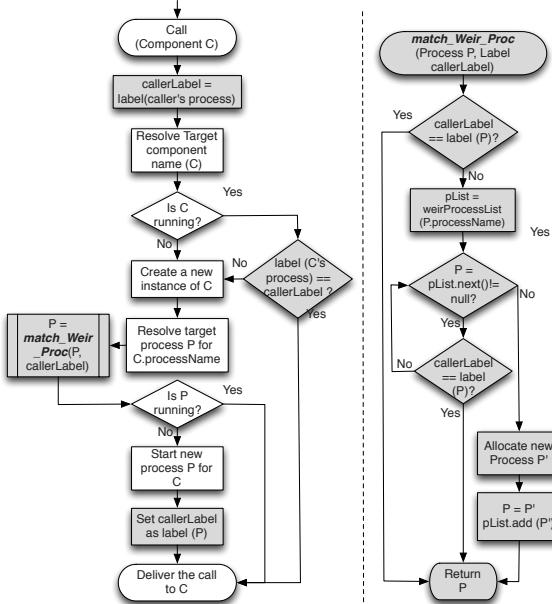


Figure 8: Flow of the Activity Manager starting a component. The areas modified or added by *Weir* are shaded.

- [44] VANDEBOGART, S., EFSTATHOPOULOS, P., KOHLER, E., KROHN, M., FREY, C., ZIEGLER, D., KAASHOEK, F., MORRIS, R., AND MAZIÈRES, D. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems (TOCS)* (2007).
- [45] WALSH, D. SELinux/OverlayFS integration. <https://twitter.com/rhatdan/status/588338475084029953>, 2015.
- [46] XU, Y., AND WITCHEL, E. Maxoid: transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM.
- [47] YANG, J., HANCE, T., AUSTIN, T. H., SOLAR-LEZAMA, A., FLANAGAN, C., AND CHONG, S. End-To-End Policy-Agnostic Security for Database-Backed Applications. *arXiv preprint arXiv:1507.03513* (2015).
- [48] YANG, J., YESSENOV, K., AND SOLAR-LEZAMA, A. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2012).
- [49] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation* (2006).
- [50] ZELDOVICH, N., BOYD-WICKIZER, S., AND MAZIERES, D. Securing Distributed Systems with Information Flow Control. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (2008).

A Component Polyinstantiation Logic

In this section, we describe *Weir*'s changes to the Activity Manager service's component and process assignment logic. Figure 8 shows the workflow inside the Activity Manager when a component *C* is called. The

shaded blocks form *Weir*'s label checks and polyinstantiation logic. Note that the figure portrays the high level steps followed by the Activity Manager, common to all components. When a call arrives, *Weir* first gets the label for the caller's process from the kernel and stores it in *callerLabel*. The Activity Manager then resolves the target component *C* using the information in the call. At this point the Activity Manager only knows the name and type of the target component (e.g., the content provider *C*). The Activity Manager then checks if there is a runtime instance of *C* in its records. If a runtime instance exists and is executing in a process with a matching label, the call is delivered to the running instance. Otherwise, *Weir* forces the Activity Manager to create another runtime instance, for this new *callerLabel*.

Without *Weir*, the Activity Manager would always deliver the call to the existing instance.⁵ *Weir* modifies the Activity Manager's internal bookkeeping structures to be consistent with its polyinstantiation; i.e., it enables the Activity Manager to manage multiple runtime records for the same component. For example, the ActivityManager uses a direct mapping between a service's name and its runtime instance, to store records of running services. *Weir* modifies this mapping to one between the name and a set of services.

At this stage, the system has a new component instance that needs to be executed in a process. The Activity Manager selects the process based on the *processName* extracted from the “android:process” manifest attribute. A runtime record of the resolved process *P* is then sent to *Weir* for process matching (i.e., the *Match_Weir_Proc (P, callerLabel)* subroutine). *Weir* first checks the label of the existing process *P*, and if it matches, returns *P* itself. If not, *Weir* retrieves its internal list of processes associated with *P*. This list constitutes the processes that were created in the past to be assigned instead of *P* for specific caller labels. *Weir* checks if the list contains a process with a label matching the current *callerLabel*; this step ensures that components with the same *processName* as well as *callerLabel* are executed in the same process. If *Weir* fails to find a matching process in the list, it allocates a new process for the *callerLabel*, and adds it to the list of existing processes mapped to the specific *processName*. This process is then returned as *P* to the Activity Manager. The Activity Manager then starts *P*, if it is not already started, and *Weir* sets its label in the kernel. Note that if the process is already started (i.e., the original *P* was matching, or a matching process was found in *Weir*'s *pList (P.processName)*, the Activity Manager does not restart it. Finally, the component instance is executed in the assigned process, and the call is delivered to it.

⁵Except in the case of standard and multi-process activities.

Screen After Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images

Brendan Saltaformaggio¹, Rohit Bhatia¹, Xiangyu Zhang¹, Dongyan Xu¹, Golden G. Richard III²

¹*Department of Computer Science and CERIAS, Purdue University
{bsaltafo, bhatia13, xyzhang, dxu}@cs.purdue.edu*

²*Department of Computer Science, University of New Orleans
golden@cs.uno.edu*

Abstract

Smartphones are increasingly involved in cyber and real world crime investigations. In this paper, we demonstrate a powerful smartphone memory forensics technique, called RetroScope, which recovers multiple previous screens of an Android app — in the order they were displayed — from the phone’s memory image. Different from traditional memory forensics, RetroScope enables *spatial-temporal* forensics, revealing the progression of the phone user’s interactions with the app (e.g., a banking transaction, online chat, or document editing session). RetroScope achieves near perfect accuracy in both the recreation and ordering of reconstructed screens. Further, RetroScope is app-agnostic, requiring no knowledge about an app’s internal data definitions or rendering logic. RetroScope is inspired by the observations that (1) app-internal data on previous screens exists much longer in memory than the GUI data structures that “package” them and (2) each app is able to perform context-free re-drawing of its screens upon command from the Android framework. Based on these, RetroScope employs a novel interleaved re-execution engine to selectively reanimate an app’s screen redrawing functionality *from within a memory image*. Our evaluation shows that RetroScope is able to recover full temporally-ordered sets of screens (each with 3 to 11 screens) for a variety of popular apps on a number of different Android devices.

1 Introduction

As smartphones become more pervasive in society, they are also increasingly involved in cyber and real world crimes. Among the many types of evidence held by a phone, an app’s prior screen displays may be the most intuitive and valuable to an investigation — revealing the intent, targets, actions, and other contextual evidence of a crime. In this paper, we demonstrate a powerful forensics capability for Android phones: recovering *mul-*

tiple previous screens displayed by each app from the phone’s memory image. Different from traditional memory forensics, this capability enables *spatial-temporal* forensics by revealing what the app displayed over a time interval, instead of a single time instance. For example, investigators will be able to recover the multiple screens of a banking transaction, deleted messages from an online chat, and even a suspect’s actions before logging out of an app.

Our previous effort in memory forensics, GUITAR [35], provides a related (but less powerful) capability: recovering the most recent GUI display of an Android app from a memory image. We call this GUI display *Screen 0*. Unfortunately, GUITAR is *not* able to reconstruct the app’s previous screens, which we call *Screens -1, -2, -3...* to reflect their reverse temporal order. For example, if the user has logged out of an app before the phone’s memory image is captured, GUITAR will only be able to recover the “log out” screen, which is far less informative than the previous screens showing the actual app activities and their progression.

To address this limitation, we present a novel spatial-temporal solution, called RetroScope, to reconstruct an Android app’s previous GUI screens (i.e., Screens 0, -1, -2... -N, N > 0). RetroScope is *app-agnostic* and does not require any app-specific knowledge (i.e., data structure definitions and rendering logic). More importantly, RetroScope achieves near perfect accuracy in terms of (1) reconstructed screen display and (2) temporal order of the reconstructed screens. To achieve these properties, RetroScope overcomes significant challenges. As indicated in [35], GUI data structures created for previous screens get overwritten almost completely, as soon as a new screen is rendered. This is exactly why GUITAR is unable to reconstruct Screen -i (i > 0), as it cannot find GUI data structures belonging to the previous screens. In other words, GUITAR is capable of “spatial” — but not “spatial-temporal” — GUI reconstruction. This limitation motivated us to seek a fundamentally different ap-

proach for RetroScope.

During our research, we noticed that although the GUI data structures for app screens dissolve quickly, the actual *app-internal data* displayed on those screens (e.g., chat texts, account balances, photos) have a much longer lifespan. Section 2 presents our profiling results to demonstrate this observation. However, if we follow the traditional memory forensics methodology of searching for [16, 25, 26, 41] and rendering [35–37] instances of those app data, our solution would require app-specific data structure definitions and rendering logic, breaking the highly desirable app-agnostic property.

We then turned our attention to the (app-agnostic) display mechanism supplied by the Android framework, which revealed the most critical (and interesting) idea behind RetroScope. A smartphone displays the screen of one app at a time; hence the apps’ screens are frequently switched in and out of the device’s display, following the user’s actions. Further, when the app is brought back to the foreground, its entire screen must be redrawn from scratch: by first “repackaging” the app’s internal data to be displayed into GUI data structures, and then rendering the GUI data structures according to their layout on the screen. Now, recall that the “old” app-internal data (displayed on previous screens) are still in memory. Therefore, we propose redirecting Android’s “draw-from-scratch” mechanism to those old app data. Intuitively, this would cause the previous screens to be rebuilt and rendered. This turns out to be both feasible and highly effective, thus enabling the development of RetroScope.

Based on the observations above, RetroScope is designed to trigger the re-execution of an app’s screen-drawing code in-place within a memory image — a process we call *selective reanimation*. During selective reanimation, the app’s data and drawing code from the memory image are logically interleaved with a live *symbiont app*, using our *interleaved re-execution engine* and *state interleaving finite automata* (Section 3.2). This allows RetroScope (within a live Android environment) to issue standard GUI redrawing commands to the interleaved execution of the target app, until the app has redrawn all different (previous) screens that its internal data can support. In this way, RetroScope acts as a “puppeteer,” steering the app’s code and data (the “puppet”) to reproduce its previous screens.

We have performed extensive evaluation of RetroScope, using memory snapshots from 15 widely used Android apps on three commercially available phones. For each of these apps, RetroScope accurately recovered multiple (ranging from 3 to 11) previous screens. Our results show that RetroScope-recovered app screens provide clear spatial-temporal evidence of a phone’s activities with high accuracy (only missing 2 of 256 re-

coverable screens) and efficiency (10 minutes on average to recover all screens for an app). We have open-sourced RetroScope¹ to encourage reproduction of our results and further research into this new memory forensics paradigm.

2 Problem and Opportunity

Different from typical desktop applications, frequent user interactions with Android apps require their screen display to be highly dynamic. For example, nearly all user interactions (e.g., clicking the “Compose Email” button on the Inbox screen) and asynchronous notifications (e.g., a pop-up for a newly received text message) lead to drawing an entirely new screen. Despite such frequent screen changes, an earlier study [35] shows that every newly rendered app screen destroys and overwrites the GUI data structures of the previous screen.

This observation however, seems counter-intuitive as Android apps are able to very quickly render a screen that is similar or identical to a previous screen. For example, consider how seamlessly a messenger app returns to the “Recent Conversations” screen after sending a new message. Given that the previous screen’s data structures have been destroyed, the app must be able to *recreate* GUI data structures for the new screen. More importantly, we conjecture that the raw, app-internal data (e.g., chat texts, dates/times, and photos) displayed on previous screens must exist in memory long after their corresponding GUI data structures are lost.

To confirm our conjecture about the life spans of (1) GUI data structures (short) and (2) app-internal data (long), we performed a profiling study on a variety of popular Android apps (those in Section 4). Via instrumentation, we tracked the allocation and destruction (i.e., overwriting) of the two types of data following multiple screen changes of each app. Figure 1 presents our findings for TextSecure (also known as Signal Messenger). It is evident that the creation of every new screen causes the destruction of the previous screen’s GUI data, whereas the app-internal data not only persists but accumulates with every new screen. We observed this trend across all evaluated apps.

Considering that a memory image reflects the memory’s content at one time instance, Figure 1 illustrates a limitation of existing memory forensics techniques (background on memory image acquisition can be found in Appendix A). Specifically, given the memory image taken after Screen 0 is rendered (as marked in Figure 1), our GUITAR technique [35] will only have access to the GUI data for Screen 0. Meanwhile, the app’s internal

¹RetroScope is available online, along with a demo video, at: <https://github.com/ProjectRetroScope/RetroScope>.

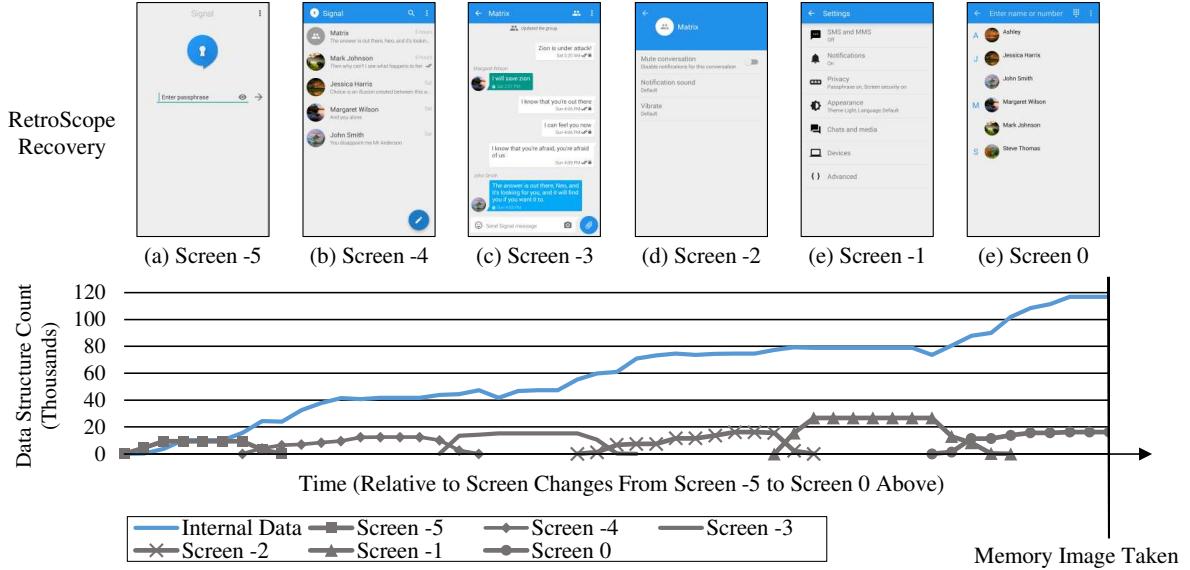


Figure 1: Life Cycles of GUI Data Structures Versus App-Internal Data Across Multiple Screen Changes.

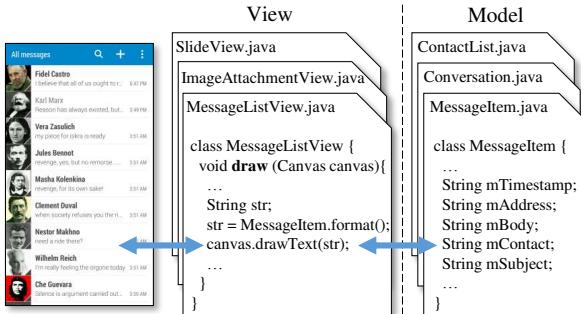


Figure 2: The Typical Model/View Implementation Split of Android Apps.

data are maintained by the app itself for as long as the app’s implementation allows (e.g., we never observed TextSecure deallocating its messages because they may be needed again). However, without app-specific data definitions or rendering logic, it is impossible for existing app-agnostic techniques [6, 36, 41] to meaningfully recover and redisplay the app’s internal data on Screens -1 to -5 in Figure 1.

It turns out that the Android framework instills the “short-lived GUI structures and long-lived app-internal data” properties in all Android apps. Specifically, Android apps must follow a “Model/View” design pattern which intentionally separates the app’s logic into *Model* and *View* components. As shown in Figure 2, an app’s Model stores its internal runtime data; whereas its View is responsible for building and rendering the GUI screens that present the data. For example, the *MessageItem*, *Conversation*, and *ContactList* (Model) classes in Figure 2 store raw, app-internal data, which are then

formatted into GUI data structures, and drawn on screen by the *MessageListView* class. This design allows the app’s View screens to respond quickly to the highly dynamic user-phone interactions, while delegating slower operations (e.g., fetching data updates from a remote server) to the background Model threads.

Further, the Android framework provides a Java class (aptly named *View*) which apps must extend in order to implement their own GUI screens. As illustrated by Figure 2’s *MessageListView* class, each of the app’s screens correspond to an app-customized *View* object and possibly many sub-*Views* drawn within the top-level *View*. Most importantly, each *View* object defines a *draw* function. *draw* functions are prohibited from performing blocking operations and may be invoked by the Android framework whenever that specific screen needs to be redrawn. This makes any screen’s GUI data (e.g., formatted text, graphics buffers, and drawing operations which build the screen) easily disposable, because the Android framework can quickly recreate them by issuing a *redraw command* to an app at any time. This design pattern provides an interesting opportunity for RetroScope, which will intercept and reuse the context of a live *redraw command* to support the reanimation of *draw* functions in a memory image.

3 Design of RetroScope

RetroScope’s operation is fully automated and only requires a memory image from the Android app being investigated (referred to as the *target app*) as input. From this memory image, RetroScope will recreate as many

previous screens as the app’s internal data (in the memory image) can support. However, without app-specific data definitions, RetroScope is unable to locate or understand such internal data. But recall from Section 2 that the Android framework can cause the app to draw its screen by issuing a redraw command, without handling the app-internal data directly. This is possible because the app’s `draw` functions are invoked in a *context-free* manner: The Android framework only supplies a buffer (called a `Canvas`) to draw the screen into, and the `draw` function obtains the app’s internal data via previously stored, global, or static variables — analogous to starting a car with a key (the redraw command) versus manually cranking the engine (app internals). Thus, RetroScope is able to leverage such commands, avoiding the low-level “dirty work” as in previous forensics/reverse engineering approaches [36, 37].

RetroScope mimics this process *within* the target app’s memory image by selectively reanimating the app’s screen drawing functions via an *interleaved re-execution engine* (IRE). RetroScope can then inject redraw commands to goad the target app into recreating its previous screens. An app’s `draw` functions are ideal for re-animation because they are (1) functionally closed, (2) defined by the Android framework (thus we know their interface definition), and (3) prevented from performing I/O or other blocking operations which would otherwise require patching system dependencies. Finally, RetroScope saves the redrawn screens in the temporal order that they were previously displayed, unless the `draw` function crashes — indicating the app-internal data could not support that screen.

To support selective reanimation, RetroScope leverages the open-source Android emulator to start, control, and modify the execution of a *symbiont app*, a minimal implementation of an Android app which will serve as a “shell” for selective reanimation.

3.1 Selective Reanimation

Before selective reanimation can begin, RetroScope must first set up enough of the target app’s runtime environment for re-executing the app’s `draw` functions. Therefore RetroScope first starts a new process in the Android emulator, which will later become the symbiont app and the IRE (Section 3.2). RetroScope then synthetically recreates a subset of the target app’s memory space from the subject memory image. Specifically, RetroScope loads the target app’s data segments (native and Java) and code segments (native C/C++ and Java code segments) back to their original addresses (Lines 1-4 of Algorithm 1) — this would allow pointers within those segments to remain valid in the symbiont app’s memory space. RetroScope then starts the symbiont app which

will initialize its native execution environment and Java runtime. Note that the IRE will not be activated until later when state interleaving (Section 3.2) is needed.

Isolating Different Runtime States. The majority of an Android app’s runtime state is maintained by its Java runtime environment². For RetroScope, it is not sufficient to simply reload the target app’s memory segments. Instead the symbiont app’s Java runtime must also be made aware of the added (target app’s) runtime data prior to selective reanimation. Later, the IRE will need to dynamically switch between the target app’s runtime state and that of the symbiont app to present each piece of interleaved execution with the proper runtime environment.

RetroScope traverses a number of global Java runtime data structures from the subject memory image with information such as known/loaded Java classes, app-specific class definitions, and garbage collection trackers (Lines 5–9 of Algorithm 1). Such data are then copied and isolated into the symbiont app’s Java runtime by inserting them (via the built in Android class-loading logic) into duplicates of the Java runtime structures in the symbiont app. Note that, at this point, the duplicate runtime data structures will not affect the execution of the symbiont app, but they must be set up during the symbiont app’s initialization so that any app-specific classes and object allocations *from the memory image* can be handled later by the IRE.

At this point, the symbiont app’s memory space contains (nearly) two full applications (shown in Figure 4). The symbiont app has been initialized naturally by the Android system with its own execution environment. In addition, RetroScope has reserved and loaded a subset of the target app’s memory segments (those required for selective reanimation) and isolated the necessary old (target app’s) Java runtime data into the new (symbiont app’s) Java runtime. The remainder of RetroScope’s operation is to (1) mark the target app’s View `draw` functions so that they can receive redraw commands and (2) reanimate those drawing functions inside the symbiont app via the IRE.

Marking Top-Level Draw Functions. RetroScope traverses the target app’s loaded classes to find top-level Views (Lines 10–17 in Algorithm 1). Top-level Views are identified as those which inherit from Android’s parent View class `ViewParent` and are not drawn inside any other Views. As described in Section 2, top-level Views are default Android classes which *contain* app-customized sub-Views. Further, we know that all Views must implement a `draw` function (which invokes the sub-Views’ `draw` functions). Thus RetroScope marks each top-level `draw` function as a reanimation starting point.

²Please see Section 5 regarding Dalvik JVM versus ART runtimes.

Algorithm 1 RetroScope Selective Reanimation.

Input: Target App Memory Image M
Output: GUI Screen Ordered Set S

```

1: for Segment  $S \in M$  do           ▷ Rebuild the Target App runtime environment
2:   if isNeededForReanimation( $S$ ) then
3:     Map( $S.startAddress$ ,  $S.length$ ,  $S.content$ )
4: SymbiontApp.initialize()          ▷ Set up Symbiont App
5: JavaGlobalStructs  $G \leftarrow \emptyset$  ▷ Isolate the Target App runtime state.
6: for Segment  $S \in M$  do           ▷ Find Java control data
7:   if containsJavaGlobals( $S$ ) then
8:      $G \leftarrow \text{getJavaGlobals}(S)$ 
9:   break
10:  ▷ Register reanimation points with the IRE.
11: View Set  $V \leftarrow \emptyset$           ▷ Top-level Views.
12: for Class  $C \in G \rightsquigarrow \text{Classes}$  do      ▷ Find top-level Views.
13:   if  $C <: \text{ViewParent}$  then        ▷ ' $<:$ ' denotes subtype.
14:     if not isSubView( $C$ ) then
15:       IRE.beginOn( $C.draw$ )          ▷ Register drawing function.
16:     View Set  $views \leftarrow C.instances$ 
17:      $V \leftarrow V \cup views$ 
18: View  $T \leftarrow \text{SymbiontApp.getTopLevelView}()$ 
19:  $T.invalidate()$                   ▷ Cause screen redraw command to be issued.
20: procedure CATCHREDRAWCOMMAND
21:   for View  $view \in V$  do
22:      $T \leftarrow view$  ▷ Override the Symbiont App's top-level View.
23:     largestID  $\leftarrow \max_{v \in view.\text{subViews}} v.getField(ID)$  ▷ Record largest subView ID.
24:     deliverRedrawCommand()        ▷ IRE handles re-execution of redrawing code.
25:   Screen  $s \leftarrow T.\text{copyGUIBuffer}()$ 
26:    $S.insert(largestID, s)$ 
27: end procedure

```

Selective Reanimation. Once all top-level draw functions are identified, RetroScope can begin selective reanimation of each. First, RetroScope invalidates the symbiont app’s current View (Line 19 of Algorithm 1). This will cause Android to set up and issue a redraw command to the symbiont app along with a buffer to draw into. However, RetroScope first intercepts this command and replaces the symbiont app’s top-level View with one of the target app’s top-level Views identified previously (Lines 20–27 in Algorithm 1). Note that RetroScope does not distinguish between different instances of top-level Views, it simply reissues redraw commands for every previously identified top-level View instance, even if duplicates exist.

Since the top-level Views of the symbiont app and the target app are both default instances of (or inherit from) the same Android View class, they are interchangeable as far as the Android framework is concerned (both with the same functionality). Now RetroScope can inject the redraw command into the symbiont app which, upon receiving this command, will naturally invoke the target app’s top-level draw function (previously marked for re-animation).

This will trigger the IRE to begin logically interleaving

the draw function execution with the symbiont app’s GUI drawing environment. Most importantly, this will direct input code/data accesses (i.e., queries to the target app’s Model) to the appropriate target app functions and output code/data accesses (i.e., drawing of screens) to the symbiont app’s running GUI framework. Upon successful completion of each draw function reanimation, RetroScope retrieves and stores the symbiont app’s (now filled) screen buffer, switches the top-level View to another marked target app View, and re-injects the redraw command — reloading the memory image in between to avoid side effects.

Finally, RetroScope reorders the redrawn screens to match the temporal order in which they were displayed. This is done via comparison of View ID fields in the target app’s Views (recovered from the memory image). A View’s ID is an integer that identifies a View. The ID may not be unique, as some Views may alias others, but it is always set from a monotonically increasing counter. This yields the property that app screens can be ordered temporally by comparing the largest ID among their sub-Views. Intuitively, the most recently modified portion of the screen (sub-View) will yield an increasingly large ID.

3.2 Interleaved Re-Execution Engine

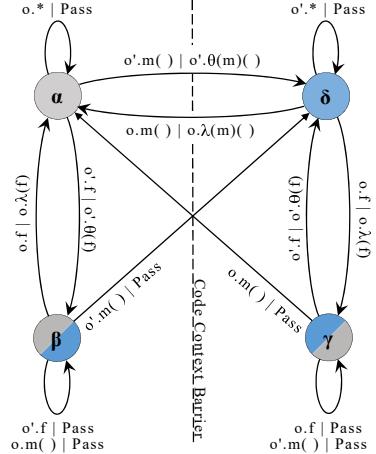


Figure 3: State Interleaving Finite Automata.

The key enabling technique behind RetroScope is its IRE which logically interleaves the state of the target app into the symbiont app just before it is needed by the execution. To monitor and interleave the execution contexts, the IRE intercepts the execution of Java byte-code instructions corresponding to function invocations, returns, and data accesses (i.e., instance/static field reads/writes). The IRE’s operation is similar to parsing a lexical context-free grammar: The current byte-code instruction (i.e., token) and the context of its operands (e.g., new/old data) are matched to a *state interleaving*

finite automata (Figure 3), where each state transition defines which runtime environment the IRE should present to that instruction.

In RetroScope, state interleaving begins at the invocation of one of the marked top-level draw functions within the target app. As a running example, Figure 4 shows a snippet of a draw function’s code along with the live memory space (containing both the symbiont app and the target app’s execution environment).

IRE State Tracking. For each byte code instruction, the IRE tracks two pieces of information: (1) if the code being executed is from the memory image (*old code*) or from the symbiont app (*new code*) and (2) if the current runtime information (i.e., loaded classes, object layouts, etc.) originates from the memory image (*old runtime*) or the symbiont app (*new runtime*). Based on that, the execution context may be in any of four possible states:

$$\begin{aligned} (\text{new code, new runtime}) &= \alpha \\ (\text{new code, old runtime}) &= \beta \\ (\text{old code, new runtime}) &= \gamma \\ (\text{old code, old runtime}) &= \delta \end{aligned} \quad (1)$$

In Figure 4, we have denoted which state the IRE is in before and after executing each line of code. For ease of explanation, Figure 4 presents source code, but RetroScope operates on byte-code instructions only. For example, before executing Line 1, the IRE is in α because no old code or data has been introduced yet. Likewise, after Line 1, the IRE is in δ as the IRE is then executing the target app’s draw function (old code) within the target app’s top-level View object (old runtime). However, note that the context of runtime data may not (and often does not) match the context of the code: For example, in Line 4, fetching the `mDensity` field from the new Canvas requires using the new runtime data but is being performed by old code (resulting in state γ).

Modeling State-Transitions. In Figure 3, we generalize the state-transition rule matching to two primitive operations: Given an object o , state transitions may occur when accessing a field f within o ($o.f$) or when invoking a method m defined by o ($o.m()$). Further, o may be an object loaded from the target app’s memory image or allocated by the target app’s code (i.e., interacting with this object requires the old runtime data), thus we denote such *old objects* as o' in Figure 3. Note that our discussion will follow Java’s object-oriented design, but the transitions in Figure 3 are equally applicable to static (i.e., $o == \text{NULL}$) execution.

The state transitions in Figure 3 are modeled as a Mealy machine [29] with the input of each state-transition being a matched operation and the output being the corresponding state correction performed by the IRE. These state corrections (i.e., transition outputs) fall into

three categories: (1) a transition from the new runtime data to the old runtime data (the function θ), (2) a transition from old to new runtime data (the function λ), and (3) no change in runtime data (“Pass”). For example, the transition from α to δ is represented as:

$$\alpha \rightarrow \delta : o'.m() \mid o'.\theta(m)() \quad (2)$$

where the input to this transition is a match on $o'.m()$ (invoking an old object’s method) and the output state correction is to switch to the old runtime prior to invoking the method ($o'.\theta(m)()$). This is exactly the IRE’s transition before executing Line 1 in Figure 4 as the IRE must switch to the old runtime prior to invoking the old View object’s draw function to look up the method’s implementation. Conversely, the transition from γ to α is represented as:

$$\gamma \rightarrow \alpha : o.m() \mid \text{Pass} \quad (3)$$

because this transition occurs when a new object’s method is invoked ($o.m$) but the IRE is already using the new runtime data, thus no runtime data correction is needed (i.e., “Pass”). This case is observed in Line 11 of Figure 4. At the beginning of Line 11, the IRE is in state γ due to the lookup of the new Canvas’s `mDensity` field on Line 4. Thus, the invocation of `getClipBounds` on Line 11 does not require the runtime to change (a “Pass” transition), but does change from old code to new.

Another important corrective action in Figure 3 is whether or not a transition crosses the code context barrier (i.e., a horizontal transition). Crossing the code context barrier signifies a switch between fetching new code (from the symbiont app) to old code (from the memory image) or vice versa. Although crossing the context barrier alone does not require active correction by the IRE (e.g., the old runtime’s method definitions will naturally direct the execution to the old code), the IRE must note that the change occurred.

Monitoring which context the code is fetched from is essential for a number of runtime checks and corrections that the IRE must perform. Firstly, objects allocated while executing old code should use the class definitions from the target app (as the Android framework classes may be vendor-customized or the class may be defined by the target app itself). Secondly, type comparisons (e.g., the Java `instanceof` operator) executed by old code must consider both new and old classes but prefer old classes. This is because new objects (which are instances of classes loaded by the symbiont app’s runtime) will be passed into old code functions — which use the target app’s loaded classes that contain “old duplicates” of classes common to both executions (e.g., system classes). The reverse is true for new code type comparisons. Lastly, exceptions thrown during interleaved execution should be catchable by both old and new code.

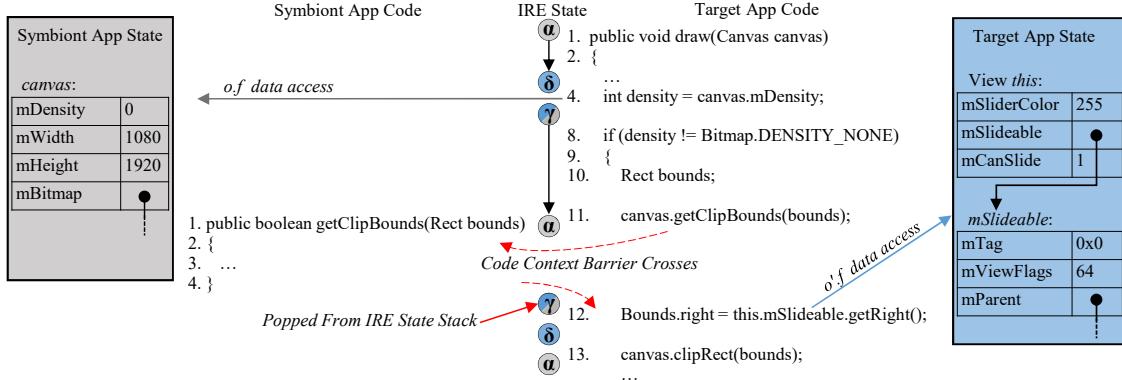


Figure 4: Example of Interleaved Re-Execution.

Interestingly, we find a number of test cases in Section 4 *purposely* throw exceptions inside their inner drawing functions, and allowing new code to catch old code exceptions (or vice versa) requires patching type lookups (as before) and stack walking.

Return Transitions. Although Figure 3 does not illustrate state transitions for return instructions, the IRE does perform state correction for them. Unlike the transitions in Figure 3 (which rely on the current IRE state to determine a new state), method returns simply restore the IRE state from before the matching invocation. This is tracked by a stack implemented in the IRE which pushes the current IRE state before invoking a method and pops/restores that IRE state upon the method’s return. This behavior is seen in Line 12 in Figure 4. Before the invocation of `getClipBounds` (Line 11), the IRE is in state γ . Function `getClipBounds` executes in state α , and upon its return the IRE pops state γ from the stack and restores that state prior to executing Line 12.

Another notable simplification of the IRE’s design is that it is sufficient to only perform state correction at function invocations, returns, and field accesses. Intuitively, this is because other “self-contained” instructions (e.g., mathematical operations) do not require support from the runtime. But another advantage is that state-interleaving tends to occur after bunches of instructions. Our evaluation shows that on average 10.24 instructions in a row will cause loop-back transitions before a state correction is needed. Further, many functions execute entirely in state α or δ because no data from the other environment enter those functions.

Native Execution. The IRE operates on the Java bytecode instructions of the functions marked for selective reanimation. However, it is possible that app developers utilize the Java Native Interface (JNI) to implement some of their app’s functionality in native C/C++ code. Further, the Android framework heavily uses JNI functions. When the IRE observes an invocation of a C/C++

function, it follows the same state transitions defined in Figure 3 (i.e., new code only invokes new C/C++ functions and vice versa).

Luckily, due to the tightly controlled interaction between C/C++ functions and the Java runtime data, the IRE’s state correction can be further simplified. To access data or invoke methods from the Java runtime, C/C++ functions must use a set of helper functions defined by the Java runtime. The IRE hooks these functions and checks if the data or method being requested is in the old or new context. The IRE can then properly patch the helper function’s return value and allow the C/C++ function to execute as intended. Note that, because all the target app’s native code and data segments have been mapped back to their original addresses, all pointers (code and data) in those segments remain valid.

Lastly, although the IRE executes app-specific code, it does so on a *syntactic* basis *without* understanding the code’s semantics, hence maintaining RetroScope’s app-agnostic property.

3.3 Escaping Execution and Data Accesses

To monitor and interleave the target app’s reanimation, the IRE must accurately track the current state of the execution environment. However, due to the relative complexity of Android apps, it is possible that the target app’s control flow causes the IRE to miss a state transition, potentially failing to correct the execution environment despite the actual execution being in a different state. We call such missed state transitions *escaping execution* or *escaping data accesses*.

Escaping Execution. This occurs when the target app’s reanimation invokes a function but the IRE is unable to determine which context to transition to. This is primarily due to the invocation of a static method which exists in both the old and new environments — leading to an *ambiguous state-transition*, where the IRE does not have

sufficient information at the function invocation site to determine which state (α) or (δ) to transition to. Simply put, the IRE must discover if the execution intended to invoke the old or new method. To decide that, the IRE performs a simple data flow analysis on each version. If the method writes data to a static variable, then the IRE always invokes the method in state α , otherwise the IRE keeps the same state that the method was invoked by (to avoid an unnecessary transition). This ensures that any accesses to static values which exist in both old and new environments are always directed to the new one. Note that app-defined static variables will only exist in the old environment, and thus their accesses do not lead to ambiguous transitions.

Escaping Data Accesses. This occurs when an app implements a non-standard means of accessing an object’s fields. For example, the two most common causes of escaping data accesses we observed are: (1) C/C++ code using a hard-coded Java object layout to access an object’s fields and (2) old Java code which has cached an old version of an object which RetroScope is trying to replace with a new version (e.g., some Views will save and reuse a reference to the previously drawn on Canvas). Although escaping data accesses are caused by app implementation differences, they can be handled uniformly by the IRE.

Escaping data accesses caused by Java code can be identified automatically when the fields of the object are accessed incorrectly. For example, there should not exist any old Canvas objects during selective reanimation and thus the IRE will identify its field accesses and replace the object with the new instance. Escaping data accesses caused by C/C++ code are handled by preventing C/C++ code from directly accessing Java objects. Instead, the IRE requires all pointers to Java objects to be encoded before they are given to C/C++ code. These pointers can be decoded when they are used in the standard JNI field access helper functions, but will cause a segmentation fault when dereferenced erroneously. This segmentation fault can then be handled by RetroScope to patch the field access with the appropriate JNI helper function. In fact, support for encoded/decoded JNI pointers already exists but may be avoided in Android, so the IRE only needs to require that all JNI pointers are encoded/decoded and handle the segmentation fault for those that previously avoided this functionality.

4 Evaluation

Evaluation Setup. Our evaluation of RetroScope involved three Android phones (a Samsung Galaxy S4,

HTC One, and LG G3)³ as evidentiary devices. On each phone, we installed and interacted with 15 different apps to cause the generation, modification, and deletion of as many screens as possible. The interactions took an average of 16 minutes per app, and we installed and interacted with the apps on each phone at random times over a 4-day period. Then, for each phone, we waited 60 minutes for any background activity of the 15 apps to complete, after which we took a memory image from the phone (as described in Appendix A).

The set of 15 apps was chosen to represent both typical app categories (to highlight RetroScope’s generic applicability) and diverse app implementation (to evaluate the robustness of RetroScope’s selective reanimation). Based on the importance of personal communication in criminal investigations, we included Gmail, Skype, WeChat, WhatsApp, TextSecure (also known as Signal, notable for its privacy-oriented design which limits evidence recovery [4]), Telegram (whose encrypted broadcast channels are popular with terrorist organizations [3]), and each device’s default MMS app (implemented by the device vendor). We also included the two most popular social networking apps: Facebook (known for its highly complex/obfuscated implementation) and Instagram. Finally we consider several apps which, by nature, display sensitive personal information: Chase Banking, IRS2Go (the official IRS mobile app), MyChart (the most popular medical record portfolio app), Microsoft Word for Android, and the vendor-specific Calendar and Contacts/Recent Calls apps.

We then used RetroScope to recreate as many previous app screens as still exist in the memory images of the 45 (15 × 3) apps. The recovery results are reported in Table 1. Table 1 presents the device and app name in Columns 1 and 2, respectively. Column 3 shows the ground-truth number of screens that RetroScope should recover, and Column 4 reports the number of screens recovered. Columns 5 through 9 present several metrics recorded over the selective reanimation of all screen redrawing functions for each app: Column 5 shows the number of reanimated Java byte-code instructions, Column 6 reports the number of JNI invocations (i.e., C/C++ functions invoked from Java code) observed, and Columns 7 and 8 report the total number of newly allocated Java objects and C/C++ structures that made up the new screens. Column 9 shows RetroScope’s runtime for each case.

Selective Reanimation Metrics. Table 1 provides interesting insights into the complexity and scale of screen redrawing via selective reanimation. From Table 1, we learn that an average of 231,867 byte-code instructions

³These devices all run vendor-customized versions of Android Kitkat (the most widely used Android version [17]).

Device	App	Expected # of Screens	RetroScope Recovery	Metrics for Evaluating Selective Reanimation				
				Byte-Code Instructions	JNI Invocations	Allocated Java Objects	New C/C++ Structures	Runtime (seconds)
Samsung S4	Calendar	8	8	259196	4699	930	79119	502
	Chase Banking	9	9	424336	9318	1905	106168	1610
	Contacts	5	5	199755	4606	928	49322	369
	Facebook	6	6	338195	7928	1432	45420	1059
	Gmail	5	5	188463	4185	826	80808	487
	Instagram	7	7	240139	5191	482	86319	672
	IRS2Go	5	5	195413	4450	790	21027	674
	MMS	3	3	96856	2004	333	25311	276
	Microsoft Word	3	3	211762	4273	460	58291	637
	MyChart	4	4	74213	1632	367	18902	259
	Skype	6	6	236213	5256	1072	30753	486
	Telegram	6	7	177973	3488	314	41815	664
	TextSecure	4	4	145436	3461	763	27450	450
	WeChat	3	3	121630	2823	638	24730	831
	WhatsApp	7	8	402536	8186	1373	65818	1390
LG G3	Calendar	7	7	199290	4193	665	72944	478
	Chase Banking	8	8	360607	8436	1843	127337	1731
	Contacts	5	5	313068	6289	1184	105004	430
	Facebook	7	7	448535	10038	1892	88949	1413
	Gmail	6	6	263850	6148	1353	239711	1248
	Instagram	5	5	245094	5097	489	104391	446
	IRS2Go	6	6	335323	7599	1458	82077	709
	MMS	6	6	147428	3077	422	61210	303
	Microsoft Word	4	4	175394	4189	652	51769	375
	MyChart	3	3	59284	1291	202	24995	335
	Skype	6	5	238227	4914	914	63007	382
	Telegram	6	6	125085	2452	183	48496	297
	TextSecure	6	6	206146	4388	860	80672	381
	WeChat	4	5	225245	5296	1293	72310	632
	WhatsApp	7	8	205661	4548	884	67789	466
HTC One	Calendar	6	6	197316	3675	732	102642	749
	Chase Banking	11	11	584587	12591	2091	266965	850
	Contacts	3	3	190847	4023	723	71578	380
	Facebook	6	5	382522	8629	1451	95516	1128
	Gmail	6	6	235973	5366	929	129804	1128
	Instagram	3	3	86829	2078	433	42037	399
	IRS2Go	5	5	200196	4510	832	52097	547
	MMS	4	4	93971	1950	287	45085	493
	Microsoft Word	3	3	137978	3249	562	43209	456
	MyChart	6	6	131876	2599	353	65377	403
	Skype	9	9	468258	9817	1232	149372	890
	Telegram	4	4	98662	1989	185	49902	291
	TextSecure	7	8	231891	5268	924	98571	488
	WeChat	5	5	211518	4836	901	69587	723
	WhatsApp	6	6	321229	7075	1571	104216	573

Table 1: Overall Results of RetroScope Evaluation.

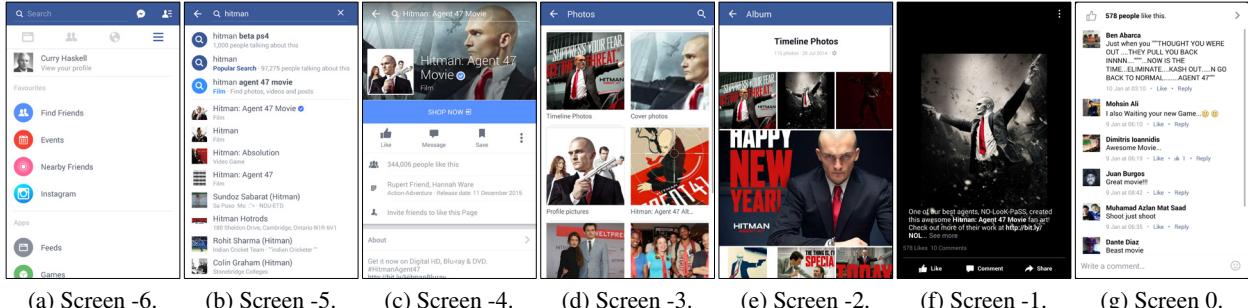
and 5,047 JNI function invocations are required to redraw all of the screens for a single app. This yields an average of 41,078 byte-code instructions and 894 JNI function invocations *per screen*. Higher than our initial expectations, these numbers attest to the complexity of the screen drawing implementation and robustness of RetroScope’s IRE.

Another metric above our expectation was the number of data structures that had to be *newly allocated* to redraw each screen. While redrawing all previous screens of each app, the reanimated code allocated an average of 891 Java objects and 76,397 C/C++ structures per app,

and an average of 158 Java objects and 13,535 C/C++ structures *per screen*. These numbers confirm the claim in GUITAR [35] that each screen is made of “thousands of GUI data structures.” Most importantly, as also shown in [35], only the structures for Screen 0 may still exist in a memory image, whereas RetroScope actively triggers the *rebuilding* of the lost data for Screens 0, -1, -2, … -N.

4.1 Spatial-Temporal Evidence Recovery

Ground Truth. We now evaluate how accurately RetroScope recreates the screens displayed during our last



(a) Screen -6. (b) Screen -5. (c) Screen -4. (d) Screen -3. (e) Screen -2. (f) Screen -1. (g) Screen 0.

Figure 5: LG G3 Facebook Recovery.

interaction session with each app. However, obtaining the ground truth (how many previous screens RetroScope *should* recover) is not straightforward because the screens’ recoverability is decided by the availability of the app’s internal data in the memory image. Therefore, to identify the recoverable previous screens, we instrumented each app to log any *non-GUI-related* data allocations and accesses performed by each screen-drawing function. We then compared this log to the content of the final memory image to identify which screens’ *entire* app-internal data still existed⁴. This gives us a strict lower bound on the number of screens that RetroScope should recover (i.e., all the internal data for those screens exist in the memory image). *Without* app-specific reverse engineering efforts, it is impossible to know the upper bound that the app’s internal data could support. But as we discuss later, screen redrawing is often “all or nothing” and adheres closely to this lower bound.

Highlights of Results. RetroScope recovered a total of 254 screens for the 45 apps, from a low of 3 to a high of 11 screens — ironically for the privacy sensitive Chase Banking app on the HTC One phone (Figure 6). Overall, Table 1 shows that RetroScope recovers an average of 5.64 screens per app, with the majority of the test cases (33 out of 45) having 5 or more screens.

Table 1 highlights the depth of *temporal* evidence that RetroScope makes available to forensic investigators, but even more intriguing is the clear progression of user-app interaction portrayed by the recovered screens. Figure 5 shows the 7 screens recovered for the Facebook app on the LG G3 phone. From these screens we can infer the “suspect’s” progression: from his own profile (Screen -6), to search results for “hitman” (Screen -5), to the Facebook profile (Screen -4), Photos screen (Screen -3), a photo album (Screen -2) of the Hitman movie, to a single photo (Screen -1), and lastly to that photo’s comments (Screen 0). Such powerful spatial-temporal recov-

ery — from a single memory image — is not possible via any existing memory forensics technique.

Another interesting observation from Table 1 is that, although RetroScope’s recovery is app-agnostic, the apps’ diverse implementations lead to very different redrawing procedures. For example, for both Skype and Facebook apps on the Samsung S4, RetroScope reproduced all 6 screens from each app. However, Facebook’s redrawing implementation appears much more complex, requiring 338,195 byte-code instructions and 7,928 JNI invocations, compared to Skype’s 236,213 byte-code instructions and 5,256 JNI invocations. This also leads to varied RetroScope run times: from the shortest, Samsung S4’s MyChart, at 259 seconds to the longest, LG G3’s Chase Banking, at 1731 seconds. The average runtime across all apps is 655 seconds (10 minutes, 55 seconds).

Lastly, Table 1 shows that in two cases (Rows 26 and 34), RetroScope missed a single screen. Manual investigation of these cases revealed that the app-specific drawing functions for the missed screens had thrown unhandled Java exceptions. For the HTC One device’s Facebook case, we found that the app had stored a pointer to the Thread object which handled its user interface and during redrawing the app failed on a check that the current Thread (handled by RetroScope during reanimation) is the same as the previously stored Thread (from the memory image). For the LG G3 Skype case, when drawing the “video call” screen, a saved timer value (in the memory image) was compared against the system’s current time, which also failed during reanimation. These were addressed by reverse engineering to determine which field/condition in the app caused the fault, and RetroScope can be instructed to set/avoid them during interleaved execution. Also of note, several cases required recovering on-screen elements (e.g., user avatars) which were cached on persistent storage until they are loaded on the screen. Currently, RetroScope attempts to detect (e.g., via the unhandled exception) but can not automatically correct such implementation-specific semantic constraints. We leave this as future work.

⁴Note that RetroScope did not have access to nor could benefit from this ground truth information. Further, we utilized in-place binary instrumentation (which does not interact nor interfere with the app’s execution or memory management) to ensure the accuracy of our experiments.

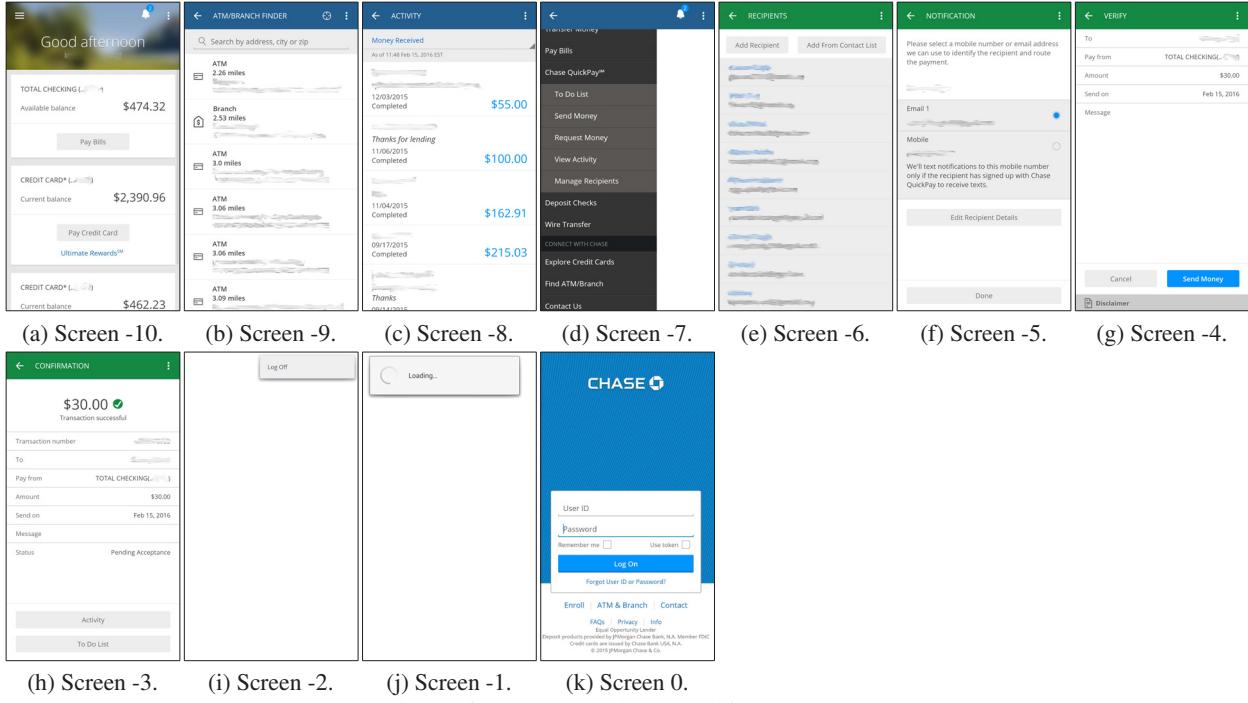


Figure 6: HTC One Chase Banking Recovery.

4.2 Case Study I: Behind the Logout

We now elaborate on the Chase Banking app case and highlight RetroScope’s ability to recreate an app’s previous screens *even after the user has logged out*. Table 1 Row 32 shows that RetroScope recovered 11 out of 11 screens (the highest of all cases). Not surprisingly, the recovery required the most reanimated byte-code instructions (584,587) and JNI function invocations (12,591), as well as the most re-allocated Java objects (2,091) and C/C++ structures (266,965).

The recovered screens are shown in Figure 6. Starting from the Account screen (Screen -10), the “suspect” looks up a nearby ATM (Screen -9). He then reviews his recent money transfers (Screen -8) and begins a new transfer to a friend via the app’s options menu (Screen -7). Screens -6 to -4 fill in the transfer’s recipient and amount. Screen -3 asks the user to confirm the transfer. Screen -2 shows the app’s “Log Out” menu, Screen -1 presents a loading screen while the app logs out, and Screen 0 is (as expected) the app’s log in screen.

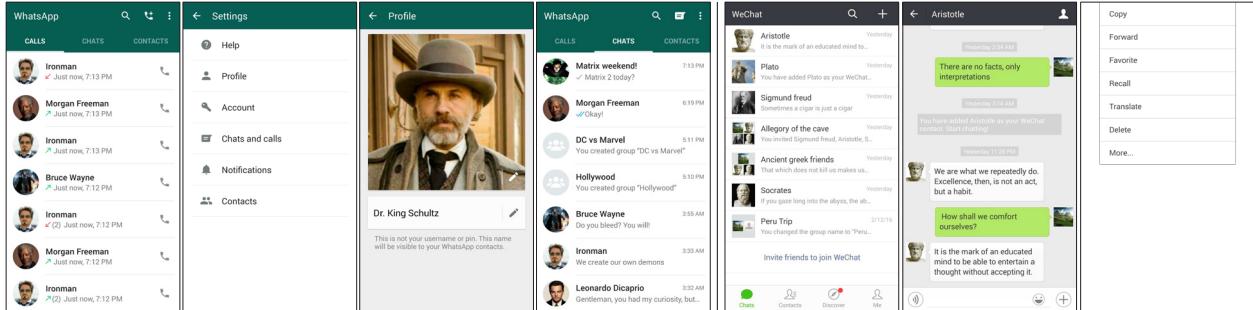
This case yields some interesting observations: First, it highlights the robustness of RetroScope to recover a large number of screens when an app’s internal data continues to accumulate. More importantly, the case shows that, after logging out, the Chase app (as well as many others we have tested) does not clear its internal data. This is not surprising because programmers usually consider their app’s *memory* to be private (compared to network communications or files on persistent storage). This is further evidenced by the TextSecure app, which

also allows for a significant post-logout recovery (of pre-logout screens), despite the app’s message database being locked in the device’s storage.

4.3 Case Study II: Background Updates

Another interesting case is WhatsApp Messenger on the Samsung S4. Table 1 Row 15 shows that RetroScope re-animated 402,536 byte-code instructions and 8,186 JNI functions in 23 minutes, 10 seconds, yielding an average of 50,317 instructions and 1,023 JNI functions per screen. What was unexpected however is that RetroScope recovered an *extra* screen (8 out of the 7 expected screens) from the memory image.

Our investigation into this extra screen found that it was not a screen we had previously seen during our phone usage. Instead, after we had finished interacting with WhatsApp, the app received a new chat message *while it was in the background* and, to our surprise, this prompted the app to prepare a new chat screen that appended the newly received message to the chat. Figure 7 presents the screens recovered by RetroScope, and again we see a clear temporal progression through the app by the “suspect.” First, Screen -6 shows the call log screen. The app’s Settings screen is seen in Screen -5 followed by a screen that is only accessible through the Settings: the device owner’s profile (our fictitious device owner is Dr. King Schultz) in Screen -4. Screen -3 shows the recent chats; Screen -2 shows the “suspect’s” chat with a friend; then Dr. Schultz places a call to that friend



(a) Screen -6.

(b) Screen -5.

(c) Screen -4.

(d) Screen -3.

(e) Screen -2.

(f) Screen -1.

(g) Screen 0.

(h) Screen +1.

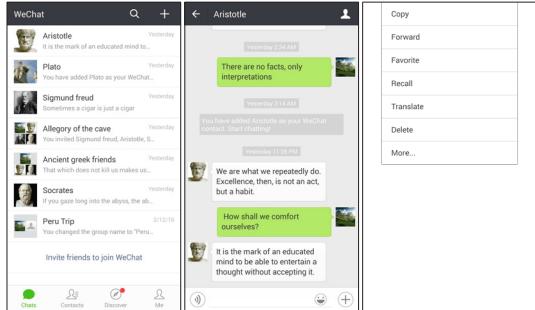
Figure 7: Samsung S4 WhatsApp Recovery.

in Screen -1. Lastly, Screen 0 shows the friend’s profile. Then, the extra *Screen +1* shows the chat screen as prepared by the app while in the background. Indeed it shows the newly received message, even time-stamped (“TODAY” and “4:51 AM” in Figure 7(h)) after the previous chat had taken place.

To ensure that this result was not an accident, we repeated the experiment (receiving chat messages while the app was in the background) six more times (twice per device). In every test, we found that RetroScope recovered the additional pre-built chat screen containing the new message. Strangely, after testing the other apps which can receive background updates, we found that WhatsApp is the only app, among our 15 apps, that exhibited this behavior. We suspect that this is a WhatsApp-specific implementation feature to speed up displaying the chat screen (*Screen +1*) when the device user clicks the “New Message” pop-up notification.

4.4 Case Study III: Deleted Messages

In addition to the WhatsApp case above, RetroScope recovered *extra screens* for four other cases in Table 1: Telegram (Row 12), WeChat (Row 29), WhatsApp (Row 30), and TextSecure (Row 43). However, the extra screens here are for a different reason: RetroScope can recover *explicitly deleted* chat messages. In these tests, we began a chat in each app and then explicitly deleted one of the messages (as a suspect would do in an attempt to hide evidence), and then used RetroScope to



(a) Screen -4.

(b) Screen -3.

(c) Screen -2.

(d) Screen -1.

(e) Screen 0.

Figure 8: LG G3 WeChat Recovery.

recover the deleted message. Additionally, RetroScope also recovered proof of the suspect’s intent to delete the message: For WeChat and WhatsApp, RetroScope recovered the app’s pop-up menu (just prior to the deleted message) which displays the “Delete Message” option. For TextSecure, RetroScope recovered both the pop-up menu and a loading screen showing the text “Deleting Messages.”

Figure 8 shows one example: RetroScope’s recovery for the WeChat app on the LG G3. Screen -4 shows the “suspect’s” recent chats followed by a chat conversation with a friend in Screen -3. Screen -2 is the pop-up menu displaying the “Delete” option. The deleted message (now disconnected from the previous chat window) is displayed in Screen -1, and the friend’s profile page (which the “suspect” navigated to last) is shown in Screen 0.

This result, in particular, highlights one of the most powerful features of RetroScope, given that it works for many apps and even provides proof of the suspect’s intent. Further, all four apps tout their *encrypted* communication and some (e.g., TextSecure) even encrypt the message database in the device. In light of this, law enforcement has routinely had trouble convincing developers of such apps to backdoor their encryption in support of investigations [4, 5]. Despite the few hardening measures discussed in Section 5, RetroScope can provide such alternative evidence which would otherwise be unavailable to investigators.

5 Privacy Implications and Discussion

RetroScope provides a powerful new capability to forensic investigators. But despite being developed to aid criminal investigations, RetroScope also raises privacy concerns. In digital forensics practice, the privacy of device users is protected by strict legal protocols and regulations [9, 21], the most important of which is the requirement to obtain a search warrant prior to performing “invasive” digital forensics such as memory image analysis. Outside the forensics context, even some of the authors were surprised by the *temporal depth* of screens that RetroScope recovered for many privacy-sensitive apps (e.g., banking, tax, and healthcare). In light of this, we discuss possible mitigation techniques which, despite their significant drawbacks, might be considered worthwhile by privacy-conscious users/developers.

RetroScope’s recovery is based on two fundamental features of Android app design: (1) All apps which present a GUI must draw that GUI through the provided View class’s `draw` function and (2) The Android framework calls drawing functions on-demand and prevents those drawing functions from performing blocking operations (file/network reads/writes, etc.). As such, an app that aims to disrupt RetroScope’s recovery would need to hinder its own ability to draw screens.

Previous *anti-memory-forensics* schemes focused on encrypting in-memory data after its immediate use. This ensures that traditional memory scanning or data structure carving approaches (e.g., [25, 26, 37, 41]) would not find any useful evidence beyond the few pieces of decrypted in-use data. However, these solutions cannot hinder RetroScope’s recovery because RetroScope recovers evidence via the app’s existing `draw` functions, which would have to include decryption routines as part of building the app screen. App developers may add state-dependent conditions to their `draw` functions which would crash when executed by RetroScope, but as seen in Section 4 these can still be handled via additional debugging/reverse engineering efforts to skip/fix the conditions.

One approach that may disable RetroScope’s recovery is to overwrite (i.e., zero) all app-internal data immediately after they are drawn on screen. By doing so, RetroScope would find that the app’s internal state could not support the execution of any of its `draw` functions. Unfortunately, this approach would significantly degrade usability and increase implementation complexity: First, frequently overwriting app-internal data would incur execution overhead (especially during screen changes which are expected to be fast and dynamic). More importantly, this would require the app to download its internal data from a remote server *every time the app needs to draw a screen*. An app may

attempt to amortize these overheads (e.g., only zeroing a prior session’s memory upon logout) but this would require: (1) tracking used/freed memory throughout the session (to be zeroed later) and (2) users to regularly log out, which is uncommon and inconvenient for frequently used apps such as email, messengers, etc.

Current vs. Future Android Runtimes. It is worth noting that Google has begun shifting the Android framework’s runtime from the Dalvik JVM to a Java-to-native compilation and native execution environment (named ART). Our implementation of RetroScope was based on the original (and still the most widely used by far [17]) Dalvik JVM runtime. However, during our development of RetroScope, specific care was taken to design RetroScope to utilize only features present in *both* runtimes. Specifically, ART still provides the same Java runtime tracking and support as Dalvik does (implemented now via C/C++ libraries) and all apps’ implementations (e.g., their `Views` and `draw` functions) remain unchanged. Our study of ART revealed that the only engineering effort required to port RetroScope is the interception of state-changing instructions in the compiled byte-code, rather than the literal byte-code as it exists in Dalvik. We leave this as future work.

6 Related Work

RetroScope is most related to GUITAR [35] which, by recovering the remaining “puzzle pieces” (GUI data structures) from a memory image, is able to piece together an app’s Screen 0. Motivated by GUITAR’s “Screen 0-only” limitation (i.e., spatial recovery), RetroScope enables the fundamentally more powerful capability of recovering Screens 0, -1, -2, ... -N (i.e., spatial-temporal recovery). Technically, GUITAR is based on geometric matching of GUI pieces; whereas RetroScope is based on selective reanimation of GUI code and data.

A number of other (spatial) memory forensics tools have also been developed recently for Android. Many of these approaches recover raw instances of app-specific data structures to reveal evidence: App-specific login credentials were recovered by Apostolopoulos et al. [8]. Macht [28] followed by Dalvik Inspector [6] involved techniques to recover Dalvik-JVM control structures and raw Java object content. Earlier, Thing et al. [42] found that text-based message contents could be recovered from memory images. Most recently, our VCR [36] technique made it possible to recover images/video/preview frames from a phone’s camera memory.

In a mobile device-agnostic effort, DEC0DE [44] involved an effective technique to carve plain-text call logs and address book entries from phone storage using probabilistic finite state machines.

RetroScope shares the philosophy of leveraging existing code for memory content rendering with our prior memory forensics technique DSCRETE [37]. However, DSCRETE renders a single application data structure, whereas RetroScope renders full app display screens in temporal order. More importantly, DSCRETE requires *application-specific* (actually, data structure-specific) identification and extraction of data rendering code, while RetroScope is totally *app-agnostic*, requiring no analysis of app-internal data or rendering logic. Finally, DSCRETE works on Linux/x86 whereas RetroScope works on the Android/ARM platform.

Many prior memory forensics techniques leverage memory image scanning and data structure signature generation approaches [11, 12, 16, 26, 32, 34, 38, 41]. Data structure signatures can be content-based [16] or “points-to” structure-based [13, 15, 25, 26, 30]. For binary programs without source code, a number of reverse engineering techniques have been proposed to infer data structure definitions [24, 27, 39]. As a fundamentally new memory forensics technique, RetroScope requires neither data structure signature generation nor memory scanning.

7 Conclusion

We have presented RetroScope, a spatial-temporal memory forensics technique (and new paradigm) that recovers multiple previous screens of an app from an Android phone’s memory image. RetroScope is based on a novel interleaved re-execution engine which selectively reanimates an app’s screen redrawing functionality without requiring any app-specific knowledge. Our evaluation results show that RetroScope can recover visually accurate, temporally ordered screens (ranging from 3 to 11 screens) for a variety of apps on three different Android phones.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by NSF under Award 1409668.

References

- [1] Advanced jtag mobile device forensics training. <http://www.teeltech.com/mobile-device-forensics-training/jtag-forensics/>, 2015.
- [2] Forensics wiki - memory imaging tools. http://forensicswiki.org/wiki/Tools:Memory_Imaging, 2015.
- [3] ISIS still using Telegram channels - Business Insider. <http://www.businessinsider.com/isis-telegram-channels-2015-11>, 2015.
- [4] Signal, the Snowden-Approved Crypto App, Comes to Android. <http://www.wired.com/2015/11/signals-snowden-approved-phone-crypto-app-comes-to-android/>, 2015.
- [5] Apple vs. the FBI: Google, WhatsApp, John McAfee and more are taking sides - LA Times. <http://www.latimes.com/business/technology/la-fi-tn-tech-response-apple-20160218-snap-htmlstory.html>, 2016.
- [6] 504ENSICS LABS. Dalvik Inspector. <http://www.504ensics.com/automated-volatility-plugin-generation-with-dalvik-inspector/>, 2013.
- [7] 504ENSICS LABS. LiME Linux Memory Extractor. <https://github.com/504ensicsLabs/LiME>, 2013.
- [8] APOSTOLOPOULOS, D., MARINAKIS, G., NTANTOGIAN, C., AND XENAKIS, C. Discovering authentication credentials in volatile memory of android mobile devices. In *Collaborative, Trusted and Privacy-Aware e/m-Services*. 2013.
- [9] ASHCROFT, J., DANIELS, D. J., AND HART, S. V. Forensic examination of digital evidence: A guide for law enforcement. *U.S. National Institute of Justice, Office of Justice Programs, NIJ Special Report NCJ 199408* (2004).
- [10] BECHER, M., DORNSEIF, M., AND KLEIN, C. Firewire: all your memory are belong to us. *CanSecWest* (2005).
- [11] BETZ, C. Memparser forensics tool. <http://www.dfrws.org/2005/challenge/memparser.shtml>, 2005.
- [12] BUGCHECK, C. Grepexec: Grepping executive objects from pool memory. In *Proc. Digital Forensic Research Workshop* (2006).
- [13] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proc. CCS* (2009).
- [14] CARRIER, B. D., AND GRAND, J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation 1* (2004).
- [15] CASE, A., CRISTINA, A., MARZIALE, L., RICHARD, G. G., AND ROUSSEV, V. FACE: Automated digital evidence discovery and correlation. *Digital Investigation 5* (2008).
- [16] DOLAN-GAVITT, B., SRIVASTAVA, A., TRAYNOR, P., AND GIFFIN, J. Robust signatures for kernel data structures. In *Proc. CCS* (2009).
- [17] GOOGLE, INC. Android dashboards - platform versions. <https://developer.android.com/about/dashboards/index.html>, 2015.
- [18] GRUHN, M. Windows nt pagefile. sys virtual memory analysis. In *Proc. IT Security Incident Management & IT Forensics (IMF)* (2015).
- [19] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: cold-boot attacks on encryption keys. In *Proc. USENIX Security* (2008).
- [20] HILGERS, C., MACHT, H., MULLER, T., AND SPREITZENBARTH, M. Post-mortem memory analysis of cold-booted android devices. In *Proc. IT Security Incident Management & IT Forensics (IMF)* (2014).
- [21] JARRETT, H. M., BAILIE, M. W., HAGEN, E., AND JUDISH, N. Searching and seizing computers and obtaining electronic evidence in criminal investigations. *U.S. Department of Justice, Computer Crime and Intellectual Property Section Criminal Division* (2009).
- [22] KOLLÁR, I. Forensic ram dump image analyser. *Master’s Thesis, Charles University in Prague* (2010).

- [23] KORNBLOM, J. D. Using every part of the buffalo in windows memory analysis. *Digital Investigation* 4 (2007).
- [24] LEE, J., AVGERINOS, T., AND BRUMLEY, D. TIE: Principled reverse engineering of types in binary programs. In *Proc. NDSS* (2011).
- [25] LIN, Z., RHEE, J., WU, C., ZHANG, X., AND XU, D. DIM-SUM: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. NDSS* (2012).
- [26] LIN, Z., RHEE, J., ZHANG, X., XU, D., AND JIANG, X. Sig-Graph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proc. NDSS* (2011).
- [27] LIN, Z., ZHANG, X., AND XU, D. Automatic reverse engineering of data structures from binary execution. In *Proc. NDSS* (2010).
- [28] MACHT, H. Live memory forensics on android with volatility. *Friedrich-Alexander University Erlangen-Nuremberg* (2013).
- [29] MEALY, G. H. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal* 34, 5 (1955), 1045–1079.
- [30] MOVALL, P., NELSON, W., AND WETZSTEIN, S. Linux physical memory analysis. In *Proc. USENIX Annual Technical Conference, FREENIX Track* (2005).
- [31] PETRONI, N., FRASER, T., MOLINA, J., AND ARBAUGH, W. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. USENIX Security* (2004).
- [32] PETRONI JR, N. L., WALTERS, A., FRASER, T., AND ARBAUGH, W. A. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation* 3 (2006).
- [33] RICHARD, G. G., AND CASE, A. In lieu of swap: Analyzing compressed ram in mac os x and linux. *Digital Investigation* 11 (2014).
- [34] SALTAFORMAGGIO, B. Forensic carving of wireless network information from the android linux kernel. *University of New Orleans* (2012).
- [35] SALTAFORMAGGIO, B., BHATIA, R., GU, Z., ZHANG, X., AND XU, D. GUITAR: Piecing together android app GUIs from memory images. In *Proc. CCS* (2015).
- [36] SALTAFORMAGGIO, B., BHATIA, R., GU, Z., ZHANG, X., AND XU, D. VCR: App-agnostic recovery of photographic evidence from android device memory images. In *Proc. CCS* (2015).
- [37] SALTAFORMAGGIO, B., GU, Z., ZHANG, X., AND XU, D. DISCRETE: Automatic rendering of forensic information from memory images via application logic reuse. In *Proc. USENIX Security* (2014).
- [38] SCHUSTER, A. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation* 3 (2006).
- [39] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: A dynamic excavator for reverse engineering data structures. In *Proc. NDSS* (2011).
- [40] SUN, H., SUN, K., WANG, Y., JING, J., AND JAJODIA, S. Trustdump: Reliable memory acquisition on smartphones. In *Proc. European Symposium on Research in Computer Security*. 2014.
- [41] THE VOLATILITY FRAMEWORK. <https://www.volatilesystems.com/default/volatility>.
- [42] THING, V. L., NG, K.-Y., AND CHANG, E.-C. Live memory forensics of mobile phones. *Digital Investigation* 7 (2010).
- [43] VIDAS, T. Volatile memory acquisition via warm boot memory survivability. In *Proc. Hawaii International Conference on System Sciences* (2010).
- [44] WALLS, R., LEVINE, B. N., AND LEARNED-MILLER, E. G. Forensic triage for mobile phones with DEC0DE. In *Proc. USENIX Security* (2011).
- [45] YANG, S. J., CHOI, J. H., KIM, K. B., AND CHANG, T. New acquisition method based on firmware update protocols for android smartphones. *Digital Investigation* 14 (2015).

Appendix

A. Memory Image Acquisition

A prerequisite of memory forensics is the timely acquisition of a memory image from the subject device. Memory images typically contain a byte-for-byte copy of the entire physical RAM of a device or the virtual memory of an operating system or specific process(es). Traditionally, acquisition is performed by investigators, before the subject device is powered down, using minimally invasive software (e.g., fmem [22], LiME [7]) or hardware (e.g., Tibble [14], CoPilot [31]) tools. Other notable techniques have used the DMA-capable Firewire port [10] to acquire memory images, existing hibernation or swap files [18, 23, 32, 33], or cold/warm booted devices [19, 20, 43], but such approaches are only employed for highly specialized investigations. A more comprehensive list of memory image acquisition tools can be found in [2].

Android memory forensics was initially proposed during the development of memory acquisition tools for the devices. Most known among these are the software-based LiME [7] and TrustDump [40] techniques. In an alternative approach, Hilgers et al. [20] proposed cold-booting Android phones to perform memory forensics. Our evaluation of RetroScope used both LiME and a ptrace-based tool we developed (also available with the open source RetroScope code). Meanwhile, hardware-based memory acquisition from a mobile device is often performed via the ARM processor's JTAG port [1, 45].

Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis

Youssra Aafer, Xiao Zhang, and Wenliang Du

Syracuse University

{yaafer, xzhang35, wedu}@syr.edu

Abstract

Android customization offers substantially different experiences and rich functionalities to users. Every party in the customization chain, such as vendors and carriers, modify the OS and the pre-installed apps to tailor their devices for a variety of models, regions, and custom services. However, these modifications do not come at no cost. Several existing studies demonstrate that modifying security configurations during the customization brings in critical security vulnerabilities. Albeit these serious consequences, little has been done to systematically study how Android customization can lead to security problems, and how severe the situation is. In this work, we systematically identified security features that, if altered during the customization, can introduce potential risks. We conducted a large scale differential analysis on 591 custom images to detect inconsistent security features. Our results show that these discrepancies are indeed prevalent among our collected images. We have further identified several risky patterns that warrant further investigation. We have designed attacks on real devices and confirmed that these inconsistencies can indeed lead to actual security breaches.

1 Introduction

When vendors, such as Samsung, LG and HTC, put Android AOSP OS on their devices, they usually conduct extensive customization on the system. The reasons for customization can be many, including adding new functionalities, adding new system apps, tailoring the device for different models (e.g., phone or tablet), or carriers (e.g., T-mobile and AT&T), etc. Further complicating the process is Android updates pushed to the devices: the updates might target a new Android or app version.

This fragmented eco-system brings in several security risks when vendors change the functionalities and configurations without a comprehensive understanding

of their implications. Previous work has demonstrated some aspects of these changes and the resulting risks. Wu et al. [25] analyze several stock Android images from different vendors, and assess security issues that may be introduced by vendor customization. Their results show that customization is responsible for a number of security problems ranging from over-privileged to buggy system apps that can be exploited to mount permission re-delegation or content leaks attacks. Harehunter [5] reveals a new category of Android vulnerabilities, called Hares, caused by the customization process. Hares occur when an attribute is used on a device but the party defining it has been removed during the customization. A malicious app can “impersonate” the missing attribute to launch privilege escalation, information leakage and phishing attacks. ADDICTED [29] finds that many custom Android devices do not properly protect Linux device drivers, exposing them to illegitimate parties.

All the problems reported so far on Android customization are mainly caused by vendors’ altering of critical configurations. They change security configurations of system apps and Linux device drivers; they also remove, add, and alter system apps. Although the existing work has studied several aspects of security problems in the changes of system/app configurations, there is no work that systematically finds all security configuration changes caused by vendor customization, how likely it can lead to security problems, what risky configuration changes are often made by vendors, etc.

In this work, we make the first attempt to systematically detect security configuration changes introduced by parties in the customization chain. Our key intuition is that through comparing a custom device to similar devices from other vendors, carriers, and regions, or through comparing different OS versions, we might be able to find security configuration changes created unintentionally during the customization. More importantly, through a systematic study, we may be able to find valuable insights in vendor customization that can help ven-

dors improve the security of their future customizations. We propose DroidDiff, a tool that detects inconsistent security configurations in a large scale, and that can be employed by vendors to locate risky configurations.

The first challenge that we face in our systematic study is to identify what configurations are security relevant and are likely to be customized. We start from the Android layered architecture and list access control checks employed at each layer. Then, for each check, we rely on Android documentation and our domain knowledge to define corresponding security features. We further analyze how different configurations of these features across custom images can lead to inconsistencies and thus affect the access control check semantics. As a result, we have identified five categories of features. DroidDiff then extracts these features from 591 custom Android ROMs that we collected from multiple sources. This step produces the raw data that will be used for our analysis.

The next challenge is how to compare these images to find out whether they have inconsistent values for the features that we extracted. Given a set of images, conducting the comparison itself is not difficult; the difficulty is to decide the set of images for comparison. If we simply compare all the 591 images, it will not provide much insight, because it will be hard to interpret the implications of detected inconsistencies. To gain useful insights, we need to select a meaningful set of images for each comparison. Based on our hypothesis that inconsistencies can be introduced by vendors, device models, regions, carriers, and OS versions, we developed five differential analysis algorithms: *Cross-Vendor*, *Cross-Model*, *Cross-Region*, *Cross-Carrier*, and *Cross-Version* analyses, each targeting to uncover inconsistencies caused by customization of different purposes. For example, in the *Cross-Vendor* analysis, we aim to know how many inconsistencies are there among different vendors; in the *Cross-Model* analysis, we attempt to identify whether vendors may further introduce inconsistencies when they customize Android for different models (e.g. Samsung S4, S5, S6 Edge).

DroidDiff results reveal that indeed the customization process leads to many inconsistencies among security features, ranging from altering the protection levels of permissions, removing protected broadcasts definitions, changing the requirement for obtaining critical GIDs, and altering the protection configuration of app components. We present our discoveries in the paper to show the inconsistency situations among each category of features and how versions, vendors, models, region, and carriers customizations impact the whole situation.

Not all inconsistencies are dangerous, but some changes patterns are definitely risky and warrant further investigation. We have identified such risky patterns, and presented results to show how prevalent they are in

the customization process. The inconsistencies expose systems to potential attacks, but if the vendors understand fully the implication of such customization, they will more likely remedy the introduced risks by putting proper protection at some other places. Unfortunately, most of the inconsistencies seem to be introduced by developers who do not fully understand the security implications. Therefore, our DroidDiff can help vendors to identify the inconsistencies introduced during their customization, so they can question themselves whether they have implemented mechanisms to remedy the risks.

To demonstrate that the identified inconsistencies, if introduced by mistakes, can indeed lead to attacks, we picked few cases detected through our differential analysis, and designed proof-of-concept attacks on physical devices¹. We have identified several real attacks. To illustrate, we found that a detected inconsistency on Nexus 6 can be exploited to trigger emergency broadcasts without the required system permission and another similar one on Samsung S6 Edge allows a non-privileged app to perform a factory reset without a permission or user confirmation. Through exploiting another inconsistency on Samsung Note 2, an attacker can forge SMS messages without the SEND_SMS permission. Moreover, an inconsistency related to permission to Linux GID mapping allows apps to access the camera device driver with a normal protection level permission. We have filed security reports about the confirmed vulnerabilities to the corresponding vendors. We strongly believe that vendors, who have source code and know more about their systems, can find more attacks from our detected risky inconsistencies. We also envision that in the future, vendors can use our proposed tool and database to improve their customization process.

Contributions. The scientific contributions of this paper are summarized as the followings:

- We have systematically identified possible security features that may hold different configurations because of the Android customization process.
- We have developed five differential analysis algorithms and conducted a large-scale analysis on 591 Android OS images. Our results produce significant insights on the dangers of vendor customization.
- We have identified risky configuration inconsistencies that may have been introduced unintentionally during customization. Our results can help vendors' security analysts to conduct further investigation to confirm whether the risks of the inconsistencies are offset in the system or not. We have confirmed via our own attacks that some inconsistencies can indeed lead to actual security breaches.

¹Due to resource limitation, we could not design the attacks for all the cases identified in our analysis.

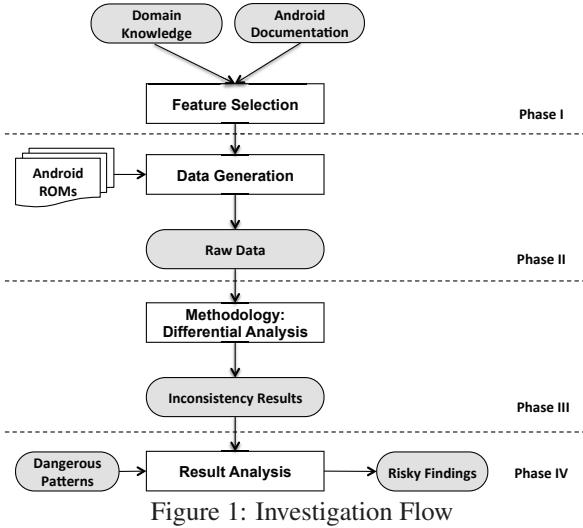


Figure 1: Investigation Flow

2 Investigation & Methodology

In this research work, we investigate Android's security features which are configurable during customization at the level of the framework and preloaded apps. Figure 1 depicts our investigation flow. As our work is data driven, the first and second phase are mainly concerned with locating and extracting meaningful security features from our collected Android custom ROMs. The two phases generate a large data set of configurations of the selected security features per image. The third phase performs differential analysis on the generated data according to our proposed algorithms to find any configuration discrepancies. It should be noted that it is out of our scope to find any security feature that is wrongly configured on all images, as obviously, it would not be detected through our differential analysis.

In the last phase, we analyze the detected discrepancies to pinpoint risky patterns. We have confirmed that they are indeed dangerous through high impact attacks. We discuss in the next sections each phase in details.

3 Feature Extraction

In this phase, we aim to extract security features that can cause potential vulnerabilities if altered inadvertently during the customization process. To systematically locate these security features, we start from the Android layered architecture (Figure 2) and study the security enforcement employed at each layer.

As Figure 2 illustrates, Android is a layered operating system, where each layer has its own tasks and responsibilities. On the top layer are preloaded apps provided by the device vendors and other third parties such as carriers. To allow app developers to access various resources and functionalities, Android Framework layer provides

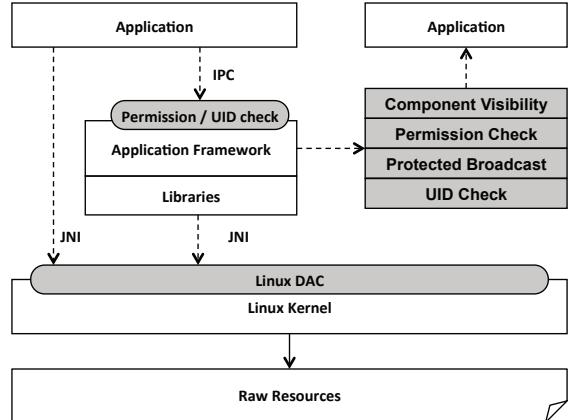


Figure 2: Android Security Model

many high-level services such as Package Manager, Activity Manager, Notification Manager and many others. These services mediate access to system resources and enforce proper access control based on the app's user id and its acquired Android permissions. Additionally, certain services might enforce access control based on the caller's package name or certificate. Right below the framework layer lies the Libraries layer, which is a set of Android specific libraries and other necessary libraries such as libc, SQLite database, media libraries, etc. Just like the framework services, certain Android specific libraries perform various access control checks based on the caller's user id and its permissions as well. At the bottom of the layers is Linux kernel which provides a level of abstraction between the device hardware and contains all essential hardware drivers like display, camera, etc. The Linux kernel layer mediates access to hardware drivers and raw resources based on the standard Discretionary Access Control (DAC).

To encourage collaboration and functionality re-use between apps, Android apps are connected together by Inter-Component Communication (ICC). An app can invoke other apps' components (e.g. activities and services) through the intent mechanism. It can further configure several security parameters to protect its resources and functionalities. As summarized in Figure 2, it can make its components private, require the caller to have certain permissions or to belong to a certain process.

Based on Figure 2, we summarize the Access Control (AC) checks employed by Android in Table 1. We specify the ones whose security features might be altered *statically* during device customization. By *static* modification, we refer to any modification that can be performed through changing framework resources files (including framework-res*.xml which contains most configurations of built-in security features), preloaded apps' manifest files and other system-wide configuration files

(platform.xml and *.xml under /etc/permissions/).

In the following section, we describe in details each configurable AC check and define its security features based on Android documentation and our domain knowledge. We further justify how inconsistent configurations of these features across custom images can bring in potential security risks. Please note that we do not discuss AC checks based on Package Names as previous work [5] has covered the effects of customizing them.

Before we proceed, we present some notations that we will be referring to in our analysis. IMG denotes a set of our collected images. E_P , E_{GID} , E_{PB} and E_C represent a set of all defined permissions, GIDs, protected broadcasts and components on IMG, respectively.

3.1 Permissions

Default and custom Android Permissions are used to protect inner components, data and functionalities. The protection level of a permission can be either Normal, Dangerous, Signature, or SystemOrSignature. These protection levels should be picked carefully depending on the resource to be protected. Signature and SystemOrSignature level permissions are used to protect the most privileged resources and will be granted only to apps signed with the same certificate as the defining app. Dangerous permissions protect private data and resources or operations affecting the user’s stored data or other apps such as reading contacts or sending SMS messages. Requesting permissions of Dangerous levels requires explicit user’s confirmation before granting them. Normal level on the other hand, is assigned to permissions protecting least privileged resources and do not require user’s approval. The following is an example of a permission declaration:

```
<permission android:name="READ_SMS"
    android:protectionLevel="Dangerous">
```

We aim to find if a permission has different protection levels across various images. For example, on vendor A, a permission READ_A is declared with Normal protection level, while on vendor B, the same permission is declared with a Signature one. This would expose the underlying components that are supposed to be protected with more privileged permissions. It would also create a big confusion for developers, as the same permission holds different semantics across images.

Formally, for each defined permission $e \in E_P$, we define the security feature f_{ne} as the following:

$$f_{ne} = \text{ProtectionLevel}(e)$$

The potential values of f_{ne} is in the set {Normal, Dangerous, Signature, Unspecified, 0}. We map

Table 1: Security Checks

AC Checks	Layer	Configurable
UID	Kernel, Framework Library, App	No
GID	Kernel	Yes
Package Name	Framework, App	Yes
Package Signature	Framework, App	No
Permission	Framework, Library, App	Yes
Protected Broadcast	App Layer	Yes
Component Visibility	App Layer	Yes
Component Protection	App Layer	Yes

SignatureOrSystem level to Signature, as both of them cannot be acquired by third party apps without a signature check. An unspecified value refers to a permission that has been defined without a protection level, while 0 refers to a permission that is not defined on an image.

3.2 GIDs

Certain lower-level Linux group IDs (GIDs) are mapped to Android permissions. Once an app process acquires these permissions, it will be assigned the mapped GID, which will be used for access control at the kernel. Permissions to GID mappings for built-in and custom permissions are defined mostly in platform.xml and other xml files under /etc/permissions/. The following is an example of a permission to GID mapping:

```
<permission android:name =
    "android.permission.NET_TUNNELING">
    <group gid="vpn" />
</permission>
```

In the above example, any process that has been granted NET_TUNNELING permission (defined with a Signature level) will be assigned the vpn GID, and consequently perform any filesystem (read, write, execute) allowed for this GID.

Android states that any change made inadvertently to platform.xml would open serious vulnerabilities. In this analysis, we aim to find if the customization parties introduce any modifications to these critical mappings and if so, what damages this might create. More specifically, we want to reveal if vendors map permissions of lower protection levels to existing privileged GIDs, which can result in downgrading their privileges. Following the same example above, assume that on a custom image, the vendor maps a permission vendor.permission (defined with Normal protection) to the existing vpn GID. This new mapping would downgrade the privilege of vpn GID on the custom image as it can be acquired with a Normal permission instead of a Signature one. Thus, any third party app granted vendor.permission will run with vpn GID

attached to its process, which basically allows it to perform any filesystem permissible for vpn GID, usually allowed to only system processes.

To allow discovering vulnerable GID to permission mappings, we extract the minimum permission requirement needed for acquiring a certain GID on a given image; i.e. the minimum protection level for all permissions mapping to it. If the same GID has different minimum requirements on 2 images, then it is potentially vulnerable. For the previous example, we should be able to reveal that vpn GID is problematic as it can be acquired with a Normal permission level on the custom image and with a Signature one on other images.

For each defined GID $e \in E_{GID}$, let P_e denote the permission set mapping to e , we define the feature f_{ne} :

$$f_{ne} = \text{GIDProtectionLevel}(e), \text{ where :}$$

$$\text{GIDProtectionLevel}(e) = \min_{p \in P_e} \text{ProtectionLevel}(p)$$

3.3 Protected Broadcasts

Protected broadcasts are broadcasts that can be sent only by system-level processes. Apps use protected broadcasts to make sure that no process, but system-level processes can trigger specific broadcast receivers. System apps can define protected broadcasts as follows:

```
<protected-broadcast android:name="broadcast.name"/>
```

Another app can use the above defined protected-broadcast through the following:

```
<receiver android:name="ReceiverA">
    <intent-filter>
        <action = "broadcast.name"/>
    </intent-filter/>
<receiver/>
```

The above ReceiverA can be triggered only by system processes broadcasting broadcast.name protected broadcast. The app can alternatively use protected broadcast through dynamically registered broadcast receivers. As it is known, during the customization process, certain packages are removed and altered. We hypothesize that because of this, certain protected broadcasts' definitions will be removed as well. We aim to uncover if these inconsistently non-protected broadcasts are still being used though, as action filters within receivers. This might open serious vulnerabilities, as the receivers that developers assumed to be only invocable by system processes will now be invocable by any third-party app and consequently expose their functionalities.

Formally, for each Protected Broadcast $e \in E_{PB}$, we define the following:

$$f_{ne} = \text{DefineUse}(e),$$

Where DefineUse(e) is defined as the following:

$$\text{DefineUse}(e) = \begin{cases} 1 & \text{if } e \text{ is used on an image but not defined} \\ 0 & \text{for other cases} \end{cases}$$

3.4 Component Visibility

Android allows developers to specify whether their declared components (activities, services, receivers and content providers) can be invoked externally from other apps. The visibility can be set through the `exported` flag in the component declaration within the app's manifest file. If this flag is not specified, the visibility will be implicitly set based on whether the component defines intent filters. If existing, the component is exported; otherwise, it is not as illustrated in the following snippet.

```
// Service1 is private to the app
<service android:name="Service1"/>
// Service2 is not private to the app
<service android:name="Service2">
    <intent-filter> ... </intent-filter>
</service>
```

We would like to uncover any component that has been exposed on one image, but not on another. We assume that if the same component name appears on similar images (e.g. same models, same OS version), then most likely, the component is providing the same functionality or protecting the same data (for content providers). Thus, its visibility should be the same across all images. To account for the cases where a component has been exported but with an added signature permission requirement, we consider them as implicitly unexposed.

Formally, for each defined component $e \in E_C$, we extract the following feature:

$$f_{ne} = \text{Exported}(e)$$

The potential values of f_{ne} is either {true, false, 0}. 0 refers to a non-existing component on a studied image.

3.5 Component Protection

Apps can use permissions to restrict the invocation of their components (services, activities, receivers). In the next code snippet, ServiceA can be invoked if the caller acquires vendor.permissionA. Moreover, an app can use permissions to restrict reading and writing to its content provider, as well as to specific paths within it. `android:readPermission` and `android:writePermission` take precedence over `android:permission` if specified, as shown in the code snippet. Components inherit their parents' permission if they do not specify one.

```
<service android:name="ServiceA"
        android:permission="vendor.permissionA"/>
```

```

<provider android:authorities="providerId"
    android:name="providerB"
    android:Permission="vendor.permissionB"
    android:readPermission="vendor.read"
    android:writePermission="vendor.write">

```

We aim to find if the same component has different protection requirements on similar images. Protection mismatch might not necessarily indicate a flaw if the component is not exposed. That's why, we only consider protection mismatches in case of exported components.

We list three cases where a component can be unintentionally exposed on one image, but protected on other images. First is the permission requirement is removed from the component's declaration. Second is the permission protecting it is of lower privilege compared to other images. Third, the permission used is not defined within the image, which makes it possible for any third-party app to define it and consequently invoke the underlying component. To discover components with conflicting protections, we map used permissions to their declarations within the same image. Any mismatch would indicate a possible security flaw for this component.

Formally, let P_e represents the permission protecting a component $e \in E_c$. We define the following feature:

$$fn_e = \text{Protection}(e);$$

Where $\text{Protection}(e)$ is defined as:

$$\text{Protection}(e) = \begin{cases} 0 & \text{if } e \text{ is not defined} \\ 1 & \text{if } P_e \text{ is None; i.e. } e \text{ is not protected} \\ \text{ProtectionLevel}(P_e) & \text{otherwise} \end{cases}$$

In the case where e is a content provider, we define P_{read} and P_{write} representing its read and write permissions and extract fn_e for both cases.

4 Data Generation

To reveal whether customization parties change the configurations of the mentioned security features, we conduct a large scale differential analysis. We collected 591 Android ROMs from Samsung Updates [4], other sources [3, 1, 2], and physical devices. These images are customized by 11 vendors, for around 135 models, 45 regions and 8 carriers. They operate Android versions from 4.1.1 to 5.1.1. Details about the collected images are in Table 2. In total, these images include on average 157 apps per image and 93169 all together apps. To extract the values of the selected security features on each image, we developed a tool called DroidDiff. For each image, DroidDiff first collects its framework resources Apks and preloaded Apks then runs Apktool to extract the corresponding manifest files. Second, it collects configuration files under /etc/permission/. Then,

Table 2: Collected Android Images

Version	# of Distinct Vendors	# of images
Jelly Bean	9	102
KitKat	9	177
Lollipop	8	312
Total	11	591

Table 3: Security Configurations Map

Image	$e \in E_p$ MIPUSH_RECEIVE	$e \in E_{GID}$ camera GID	$e \in E_c$ sms
I1: Xiaomi RedMi 1 Version: 4.4.2	Signature	Normal	True
I2: Xiaomi Mi 2A Version: 4.1.1	Unspecified	Dangerous	False

DroidDiff searches the extracted manifests and configuration files for the definitions of the targeted entities (E_p , E_{PB} , E_{GID} and E_c). Finally, DroidDiff runs the generated values through our differential analysis methodologies, discussed in the next section.

5 Differential Analysis

In our analysis, we aim to detect any feature fn_e having inconsistent values throughout a candidate set of images. Any inconsistency detected indicates a potential unintentional configuration change introduced by a customization party and requires further security analysis to assess possible consequent damages.

Let $fv(fn_e, img)$ represent the value of the feature fn_e on a given image img . To illustrate fn_e to $fv(fn_e, img)$ mappings, consider this real world example depicted in Table 3. As shown, we extract 3 security features and their corresponding values from 2 Xiaomi images. For the custom permission $e = \text{MIPUSH_RECEIVE}$, our feature extraction step generates the following values $fv(fn_e, I1) = \text{Signature}$, and $fv(fn_e, I2) = \text{Unspecified}$.

Let IMG denote a set of candidate images to be compared, we define a feature fn_e as inconsistent if:

$$\begin{aligned} C(fn_e) = \exists x \exists y [x \in IMG \wedge y \in IMG \\ \wedge x \neq y \wedge fv(fn_e, x) \neq fv(fn_e, y)] \end{aligned}$$

The above statement means that we consider the feature fn_e inconsistent across the set IMG if there exists at least two different images where the value of fn_e is not equal. It should be noted that we do not consider any cases where $fv(fn_e, img) = 0$ for $e \in \{E_p, E_{GID}$ and $E_c\}$.

Sample Selection. To discover meaningful inconsistencies through differential analysis, our collected images should be clustered based on common criteria. A meaningful inconsistency would give us insights about the responsible party that introduced it. For example,

to reveal if inconsistencies are introduced by an OS upgrade, it would not make sense to select images from all vendors, as the inconsistency could be due to customizing the device for a specific vendor, rather than because of the OS upgrade. Similarly, to uncover if a specific vendor causes inconsistencies in a new model, it is not logical to compare it with models from other vendors. Rather, we should compare it with devices from the same vendor. Besides, to avoid detecting a change caused by OS version mismatches, the new model should be compared to a model running the same OS version.

We designed five different algorithms that target to uncover meaningful inconsistencies. Specifically, by carefully going through each party within the customization chain, we designed algorithms that would reveal inconsistencies (if any) caused by each party. Further, for each algorithm, we select our candidate images based on specific criteria that serve the purpose of the algorithm,

We describe each algorithm as well as the sample selection criteria in the next sections.

A1: Cross-Version Analysis. This analysis aims to uncover any inconsistent security features caused by OS version upgrades. We select candidate image sets running similar device models to make sure that the inconsistency is purely due to OS upgrade. For instance, we would pick 2 Samsung S4 devices running 4.4.4 and 5.0.1 as a candidate image set, and would reveal if upgrading this model from 4.4.4 to 5.0.1 causes any security configuration changes. Formally, let IMG_{MODEL} denote the candidate image set as the following:

$$IMG_{MODEL} = \{img_1, img_2, \dots, img_n\}$$

such that $img_i \in IMG_{MODEL}$ if $model(img_i) = MODEL$

Based on our collected images, this algorithm generated 135 candidate image sets (count of distinct model).

Let $fv(fne, img)$ denote a value for a feature fne in $img \in IMG_{MODEL}$. We define the inconsistency condition under Cross-Version analysis algorithm as follows,

$$C_{Version}(fne) = \exists x \exists y [x \in IMG_{MODEL} \wedge y \in IMG_{MODEL} \wedge x \neq y \wedge fv(fne, x) \neq fv(fne, y) \wedge version(x) \neq version(y)]$$

The above condition implies that fne is inconsistent if there exist two same model images running different versions, and where the values of fne is not the same. Droid-Diff runs the analysis for each of the 135 candidate sets and generate the number of inconsistencies detected.

A2: Cross-Vendor Analysis. This analysis aims to reveal any feature fne that is inconsistent across vendors. To make sure that we are comparing images of similar criteria across different vendors, we pick candidate image sets running the same OS version (e.g. HTC M8 and

Nexus 6 both running 5.0.1). Our intuition here is that if an inconsistency is detected, then the vendor is the responsible party. We formally define the candidate image set as the following:

$$IMG_{VERSION} = \{img_1, img_2, \dots, img_n\}$$

such that $img_i \in IMG_{VERSION}$ if $version(img_i) = VERSION$

This algorithm generated 12 candidate image sets (count of distinct OS versions that we collected).

Let $fv(fne, img)$ denote a value for a feature fne in $img \in IMG_{VERSION}$. We redefine the inconsistency condition under Cross-Vendor analysis as follows:

$$C_{Vendor}(fne) = \exists x \exists y [x \in IMG_{VERSION} \wedge y \in IMG_{VERSION} \wedge x \neq y \wedge fv(fne, x) \neq fv(fne, y) \wedge vendor(x) \neq vendor(y)]$$

The last condition implies that fne is inconsistent if there exists two images from different vendors, but running the same OS version, where its value is not equal.

A3: Cross-Model Analysis. In this analysis, we want to uncover any feature fne that is inconsistent through different models. For example, we want to compare the configurations on Samsung S5 and Samsung S4 models, running the same OS versions. To ascertain that any inconsistency is purely due to model change within the same vendor, we pick our candidate image sets running the same OS version, defined as $IMG_{VERSION}$ in the previous example. We further make sure that we are comparing models from the same vendor by adding a new check in the next condition.

Let $fv(fne, img)$ denote a value for fne in $img \in IMG_{VERSION}$. We redefine the inconsistency condition under Cross-Model analysis as follows:

$$C_{Model}(fne) = \exists x \exists y [x \in IMG_{VERSION} \wedge y \in IMG_{VERSION} \wedge x \neq y \wedge fv(fne, x) \neq fv(fne, y) \wedge vendor(x) = vendor(y) \wedge model(y) \neq model(x)]$$

The last condition implies that fne is inconsistent if there exists two images from the same vendor, running the same OS version, but customized for different models, where its value is not equal.

A4: Cross-Carrier Analysis. We aim to uncover any inconsistent security features fne through different carriers (e.g., a MotoX from T-Mobile, versus another one from Sprint). To make sure that we are comparing images running the same OS version, we pick our candidate image sets from $IMG_{VERSION}$. We further make sure that we are comparing images running the same model

as shown in the following inconsistency condition:

$$\begin{aligned} C_{Carrier}(fne) = \exists x \exists y [x \in \text{IMGVERSION} \wedge y \in \text{IMGVERSION} \\ \wedge x \neq y \wedge fv(fne, x) \neq fv(fne, y) \\ \wedge carrier(x) \neq carrier(y) \wedge model(y) = model(x)] \end{aligned}$$

The last conditions in the above definition of $C_{Carrier}$ implies that fne is inconsistent if there exists two images running the same model and OS versions, but from different carriers where its value is not the same.

A5: Cross-Region Analysis. This analysis intends to find any inconsistencies in the configuration of security features fne through different regions (e.g. LG G4, Korean edition versus US edition). Any inconsistencies detected will be attributed to customizing a device for a specific region. We pick our candidate image sets from IMGVERSION to make sure that we are comparing images running the same OS version. We define the inconsistency count under Cross-Carrier analysis as follows:

$$\begin{aligned} C_{Region}(fne) = \exists x \exists y [x \in \text{IMGVERSION} \wedge y \in \text{IMGVERSION} \\ \wedge x \neq y \wedge fv(fne, x) \neq fv(fne, y) \\ \wedge region(x) \neq region(y) \wedge model(y) = model(x)] \end{aligned}$$

The last conditions in the above definition of C_{Region} implies that fne is inconsistent if there exists two images running the same model and OS versions, but from different regions where its value is not the same.

6 Results and Findings

We conduct a large-scale differential analysis on our collected images using the aforementioned methodologies with the help of DroidDiff. The analysis discovered a large number of discrepancies with regards to our selected features. In this section, we present the results and findings.

6.1 Overall Results

Figure 3 shows the overall changes detected from our analysis. We plot the average percentage of inconsistencies detected for each feature category using the five differential analysis algorithms. To provide an estimate of the inconsistencies count, each box plot shows an average number of total common entities (appearing on at least 2 images) in the image sets studied; we depict this number as # total in the graph. Let us use the first box plot as an example to illustrate what the data means: under the Cross-Version analysis (A1), DroidDiff generated on average 673 common permissions per each studied candidate sets. 50% of the candidate image sets contain at least 4.8% of total permissions (around 32 out of 673)

having inconsistent protection levels; those in the top 25 percentile (shown in the top whisker) have at least 6% (40) inconsistent permissions. Figure 3 also depicts the image sets that are outliers, i.e., they have particularly higher number of inconsistencies compared to the other image sets in the same group. For instance, the candidate image set $\text{IMGVersion}=4.4.2$ in the Cross-Vendor analysis (A2) contains around 10% of GIDs whose protections are inconsistent.

As depicted in Figure 3, the Cross-Version analysis (A1) detects the highest percentage of inconsistencies in all 5 categories, which means that upgrading the same device model to a different OS version introduces the highest security configuration changes. An intuitive reason behind this is that through a new OS release, Android might enforce higher protections on the corresponding entities to fix some discovered bugs (e.g. adding a permission requirement to a privileged service). However, we found out that through newer OS releases, certain security features are actually downgraded, leading to potential risks if done unintentionally. We discuss this finding in more details in Section 6.6.

Through the Cross-Vendor analysis (A2), DroidDiff detects that several security features are inconsistent among vendors, even though they are of the same OS version. We have further analyzed the vendors that cause the highest number of inconsistencies. An interesting observation is that smaller vendors, such as BLU, Xiaomi and Digiland caused several risky inconsistencies. In fact, all inconsistent GIDs are caused by these 3 companies. Probably, small vendors may not have enough expertises to fully evaluate the security implications of their actions.

The Cross-Model analysis (A3) also detects a number of inconsistencies, which means that different device models from the same vendor and OS version, might have different security configurations.

Although the Cross-Carrier (A4) and Cross-Region (A5) analyses detect a smaller percentage of inconsistencies, it is still significant to know that the same device model running the same OS version might have some different configurations if it is customized for different carriers or regions. Our results shows that the inconsistencies are less common in North America region, and more prevalent in Chinese editions.

6.2 Permissions Changes Pattern

Protection level mismatch. DroidDiff results confirm that Android permissions may hold different protection levels across similar images. As Figure 3 illustrates, more than 50% of the candidate image sets contain at least 32 (out of 673), 9 (out of 817) permissions having inconsistent protection levels in the Cross-Version (A1) and Cross-Model (A3) analyses, respectively. To reveal

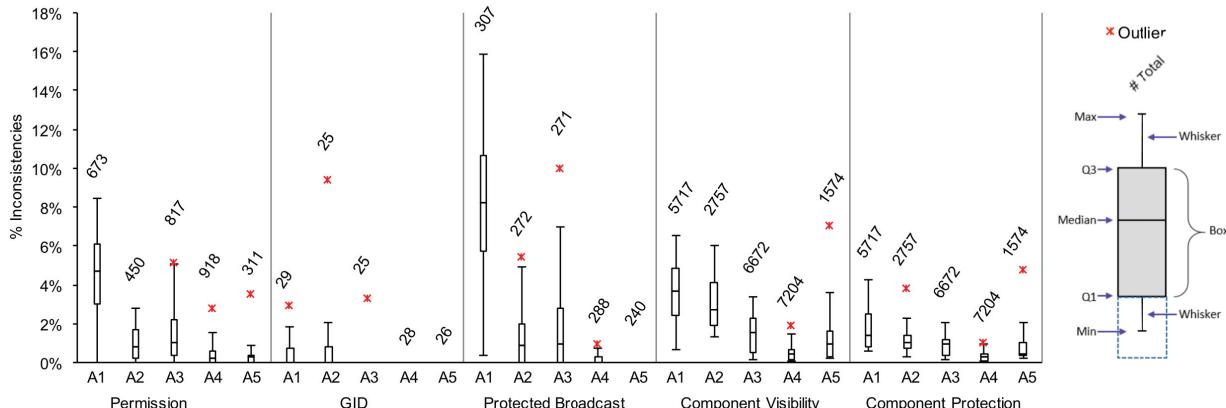


Figure 3: Overall Inconsistencies Detected

A1: Cross-Version, A2: Cross-Vendor, A3: Cross-Model, A4: Cross-Carrier, A5: Cross-Region

more insights, we checked which combination of protection level changes are the most common. That is, which combination out of the following 3 possible combinations is the most common (Normal, Dangerous), (Normal, Signature) or (Dangerous, Signature). We have calculated the occurrence of each pattern, and present the results in Figure 4. As shown, (Normal, Signature) combination is the most common pattern. This is quite serious as several permissions that hold a Signature protection level on some images are defined with a Normal protection level on others. We present here two permissions holding inconsistent protection levels:

- com.orange.permission.SIMCARD_AUTHENTICATION holds Signature and Normal protection on Samsung S4(4.2.2) and Sony Xperia C2105 (4.2.2), respectively.
- com.sec.android.app.sysscope.permission.RUN_SYSCOPE holds Dangerous and Signature protection on Samsung Note4 (5.0.1) and S4(5.0.1).

Usage of unspecified protection level. Android allows developers to define a permission without specifying a protection level, in which case, the default protection level is `Normal`. In our investigation, we found that it is not clear whether developers really intended to use `Normal` as the protection level. We found that a large percentage of these permissions (with unspecified protection level) hold conflicting protections on other images. Overall, 2% of the permissions studied were defined without a specified protection level in at least one image. To check if developers intended to use `Normal` as the protection level, for each permission that has been defined without a protection level, we check its corresponding definitions on other images to see if it has a protection level specified. We then compare the other specification to see if it is `Normal` or not. As Figure 5(a) illustrates, on average, 91% of these permissions holding

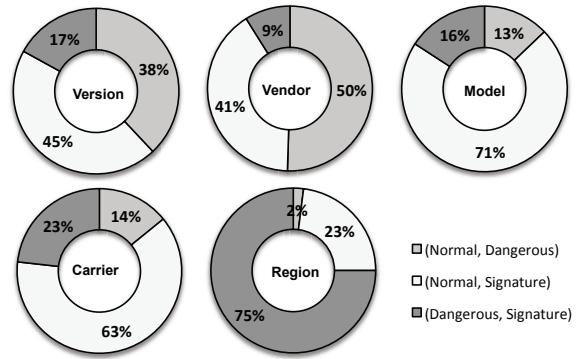


Figure 4: Protection Level Changes Patterns

unspecified protection level hold a `Signature` protection on at least 1 other image, which indicates that developers probably intended to use the `Signature` protection level. We illustrate this finding with 2 permissions:

- com.sec.android.phone.permission.UPDATE_MUTE_STATUS holds Unspecified and `Signature` protections on Samsung E7 (5.1.1) and S6 Edge(5.1.1), respectively.
- com.android.chrome.PRERENDER_URL holds Unspecified and `Signature` protections on LG Vista (4.4.2) and Nexus7 (4.4.2), respectively.

6.3 Permission-GID Mapping

By analyzing the differential analysis results of the mappings between GIDs and permissions, we have confirmed that customization introduces problematic GID-to-permission mappings that can lead to serious vulnerabilities in the victim images. Through the Cross-Vendor analysis (A2), DroidDiff detects 3 inconsistent cases (out of 25 common GIDs), in which vendors mapped less privileged permissions to privileged GIDs. This dangerous pattern leads to downgrad-

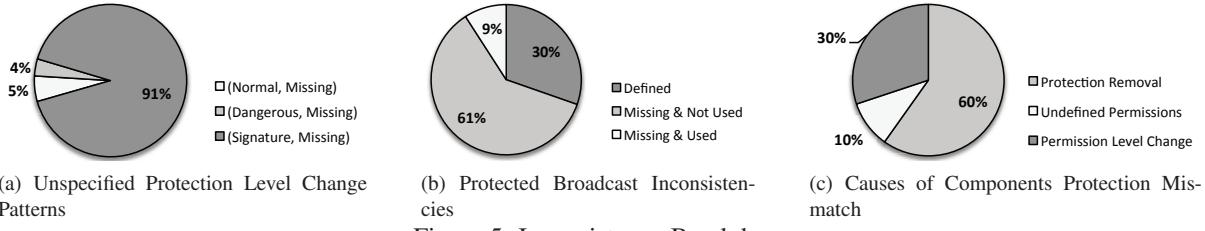


Figure 5: Inconsistency Breakdown

ing the protection level of these GIDs. We illustrate this finding with one detected example. On AOSP images and several customized images (running 4.4.4 and below), camera GID is mapped to a Dangerous level permission (`android.permission.CAMERA`). However, on Neo 4.5 (BLU), we found out that the same GID is mapped to a Normal level permission: `android.permission.ACCESS_MTK_MMHW`. This case indicates that BLU has downgraded the requirement for apps to obtain the camera GID. Our analysis reveals that the requirements for two more GIDs, system GID and media GID, have been downgraded. These two GIDs, protected by a Signature permission on most devices, can be acquired with a Normal permission on the victim devices.

6.4 Protected Broadcasts Changes Pattern

DroidDiff further reveals that protected broadcasts' definitions might be removed from some images during the customization process. As illustrated in Figure 5(b), through the Cross-Version analysis (A1), we detected that 70% of protected broadcast are not defined on at least one vendor. This might not necessarily be problematic if the broadcast is not used. However, our investigation shows that around 9% of these inconsistently unprotected broadcasts (28 on average per image set) are used as intent-filters actions for broadcast receivers. This inconsistency across versions is quite alarming as a privileged receiver that was supposed to be invoked by system processes can be invoked by any unprivileged app on certain versions. As Figure 3 further illustrates, Cross-Vendor (A2) and Cross-Model (A3) analyses reveal that more than 25% of candidate image sets contain at least 2% broadcasts which are inconsistently protected, but still being used as intent-filter actions.

6.5 Component Security Changes Pattern

Visibility mismatch. DroidDiff results confirm that app components may have a conflicting visibility. That is, the component is exposed on one image but not on another. As Figure 3 illustrates, 50% of the candidate image sets contain at least 3.9% components (around 222)

and 2% (133) holding inconsistent visibility through various versions (A1) and models (A3), respectively. To provide insights about which components hold more visibility inconsistencies, we break down our findings to activities, services, receivers, and content providers. We plot the results in Figure 6. As depicted, content providers and activities have the highest visibility mismatch. In fact, 25% of the candidate image sets contain at least 20% (53) and 14% (21) content providers holding a different visibility in different versions (A1) and vendors (A2), respectively. Similarly, 4% (139) and 3% (45) of activities hold a conflicting visibility in 50% of the studied sets based on A1 and A2, respectively.

Permission mismatch. DroidDiff further reveals that components may hold inconsistent protections across images. We break down our findings in Figure 8 (see appendix). Our results show that content providers exhibit the highest number of protection inconsistencies. In fact, more than 25% of the candidate images sets include at least 19% (51) and 10% (33) content providers having different protections in the Cross-Version (A1) and Cross-Model (A3) analyses, respectively. We have further analyzed these inconsistent components and categorized the reason behind the discrepancies. As Figure 5(c) illustrates, in the majority of the cases (60%), the discrepancy is caused by the same component being protected with a permission on one image, but not protected at all on others. The second common reason (30%) is that the same component is protected with permissions holding different protection levels across the studied images. Using non-defined permissions to protect a component is third common reason (10%).

Duplicate components declaration. Based on our analysis of the inconsistent broadcast receivers (particularly high on Lollipop images), we found out that most of them are caused by a non-safe practice that developers follow. Developers declare duplicate broadcast receivers names in the same app, but assign them different protections. After further investigation, we found out that it is not a safe practice to do as it will be possible to bypass any restrictions put on the first defined receiver. To illustrate, consider the following receivers, defined in Samsung's preloaded PhoneErrorService app:

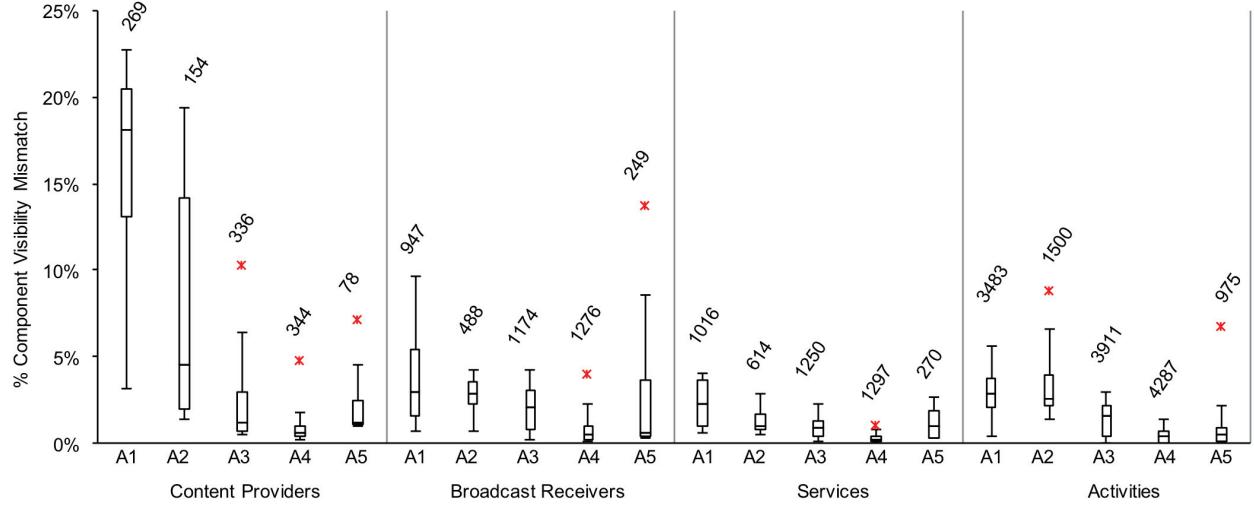


Figure 6: Breaking Down Components: Visibility Mismatch

```

<receiver android:name="PhoneErrorReceiver"
    android:permission="android.permission.REBOOT">
    <intent-filter>
        <action android:name="REFRESH_RESET_FAIL"/>
        ...
    </intent-filter>
</receiver>
<receiver android:name="PhoneErrorReceiver">
    <intent-filter>
        <action
            android:name="DATA_ROUTER_DISPLAY"/>
    </intent-filter>
</receiver>

```

In the above code, the developer decided to protect the functionality triggered when receiving the action `REFRESH_RESET_FAIL` with the permission `REBOOT` (signature level). In the other case, she decided not to require any permissions when invoking the functionality triggered by the action `DATA_ROUTER_DISPLAY`. At first glance, the above duplicate components declaration might look fine. However, we found out that the PackageManagerService does not carefully handle the registration of duplicate receivers. On one hand, it correctly handles mapping each filter to the required permission, used for **implicit** intents routing (e.g., sending the action `REFRESH_RESET_FAIL` requires `REBOOT` permission, while sending `DATA_ROUTER_DISPLAY` does not require any permission). On the other hand, however, it does not correctly map each component name to the required permission, used for **explicit** intents routing (e.g., the first `PhoneErrorReceiver` should require `REBOOT` while the second one should not). In fact, it turns out that the second declaration of the component name replaces the first one. Thus, any protection requirement on the second receiver would replace the first receiver's permission requirement in case of **explicit** invocation. Consequently, in the above example, invoking `PhoneErrorReceiver` explicitly does not require any permission. The explicit in-

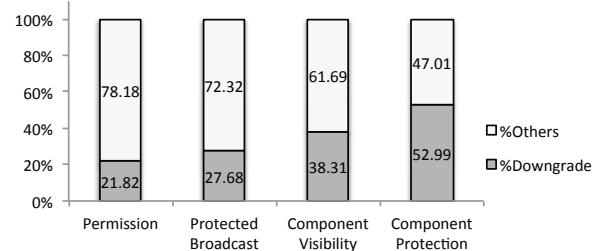


Figure 7: Percentage of Security Features Downgrades

tent can further set the action `REFRESH_RESET_FAIL` and thus trigger the privileged functionality (rebooting the phone) without the required `REBOOT` permission. We have confirmed this dangerous pattern in several preloaded apps and were able to achieve various damages. We filed a bug report about this discovered vulnerability to Android Security team and informed other vendors about it.

6.6 Downgrades Through Version Analysis

A dangerous pattern that we are interested in is whether there are any security downgrades through versions. For example, unlike a security configuration upgrade, possibly attributed to fixing discovered bugs in earlier images, downgrading a security configuration is quite dangerous as it will lead to a potential exposure of privileged resources that were already secured on previous versions. For each security configuration, we report in Figure 7, the percentage of security configuration downgrades out of all detected cases. As Figure 7 illustrates, a large number of configurations are indeed downgraded. For example, 52% of inconsistent component protection mismatch are actually caused by downgrading the protection.

7 Attacks

We would like to find out whether the risky patterns discovered can actually lead to actual vulnerabilities. To do that, we have selected some high impact cases, and tried to design attacks to verify whether these cases can become vulnerabilities. Due to resources limitations, our verification is driven by the test devices that we have, including Samsung Edge 6 Plus (5.1.1), Edge 6 (5.0.1), Nexus 6 (5.1.1), Note2 (4.4.2), Samsung S4 (5.0.1), MotoX (5.0.1), BLU Neo4 (4.2.2), and Digiland DL700D (4.4.0). We have found 10 actual attacks, some of which were confirmed on several devices. We have filed security reports for the confirmed vulnerabilities to the corresponding vendors. We discuss here 6 attacks. At the end of this section, we discuss possible impacts of 40 randomly selected cases in other devices to demonstrate the significance of inconsistent security configurations.

Stealing emails. SecEmailSync.apk is a preloaded app on most Samsung devices. It includes a content provider, called "com.samsung.android.email.otherprovider", which maintains a copy of user's emails received through the default Samsung email app. Our Cross-Model and Cross-Region analyses reveal inconsistent permission protections on this provider among several Samsung images. The Read and Write accesses to this provider are protected with a Signature permission "com.samsung.android.email.permission.ACCESS_PROVIDER" on Samsung Grand On(5.1.1, India), S6 Edge (5.1.1, UAE), and other devices. However, this provider is not protected with any permission on several other devices such as our test device S6 Edge (5.1.1, Global edition). We wrote an attack app that queries this content provider. It was able to access user's private emails on the victim device without any permission.

Forging premium SMS messages. The TeleService package (`com.android.phone`) is preloaded on many Samsung devices, and provides several services for phone and calls management. A notable service is `.TPhoneService`, which performs some major phone functionalities such as accepting voice and video calls, dialing new phone numbers, sending messages (e.g. to inform why a call cannot be received), as well as recording voice and video calls. Our Cross-Model and Cross-Version analyses reveal a permission mismatch on this critical service. On several devices, such as Samsung S5 LTE-A (4.4.2, Korea), the access to this service is protected with the Signature permission `com.skt.prod.permission.OEM_PHONE_SERVICE`, which makes the service unaccessible to third-party apps. However, on several other devices such Samsung Note 2 (4.4.2, Global edition), this service is protected with another permission

`com.skt.prod.permission.PHONE_SERVICE` for which our analysis reveals a missing definition. We built an attack app that defines the missing permission with a `Normal` protection level. Our app was able to successfully bind to `com.android.phone.TPhoneService` and invoke the send-message API on Samsung Note 2, allowing to forge SMS messages without the usually required `SEND_SMS`.

Unauthorized factory reset. The preloaded Samsung app ServiceModeApp_FB.apk performs various functionalities related to sensitive phone settings. It includes a broadcast receiver `ServiceModeAppBroadcastReceiver` that listens to several intent filters including the action filter `com.samsung.intent.action.SEC_FACTORY_RESET_WITHOUT_FACTORY_UI` that allows to factory reset the phone and delete all data without user confirmation. Our Cross-Version analysis reveals a protection mismatch for this critical broadcast receiver. In most devices running Kitkat and below, this receiver is protected with the `Signature` permission `com.sec.android.app.servicemodeapp.permission.KEYSTRING`. However, on several Lollipop images, it is not correctly protected. Further investigation reveals that this is caused by the duplicate receiver pattern discussed in Section 6.5. The declaration of the receiver has been duplicated on the victim images such that the first one requires a `Signature` permission while the second one does not. As discussed in Section 6.5, using this risky pattern allows a caller app to bypass any restrictions on the first declared broadcast receivers through explicit invocation. We wrote an attacking app that invokes the broadcast receiver explicitly with the action `com.samsung.intent.action.SEC_FACTORY_RESET_WITHOUT_FACTORY_UI` and were able to factory reset several victim devices including the latest S6 Edge Plus 5.1.1, S6 Edge 5.0.1, and S4 5.0.1.

Accessing critical drivers with a normal permission. Our Cross-Vendor analysis reveals a critical protection downgrade of the `system` GID. On some images, such as Samsung S5 (4.4.2), this GID is mapped to the `Signature` permission `com.qualcomm.permission.IZAT`. Nevertheless, on other images (e.g., Redmi Note 4.4.2 and Digiland DL700D 4.4.0), this GID is mapped to a `Normal` level permission `android.permission.ACCESS_MTK_MMHW`, indicating that any third-party app can easily get the `system` GID. Table 4 lists the device drivers that are accessible via the `system` GID on the Digiland DL700D Tablet. These are privileged drivers, but they can now be accessible to normal apps.

Triggering emergency broadcasts without permission. CellBroadcastReceiver is a preloaded Google

Table 4: Drivers accessible to System GID

Driver	ACL
booting; devmap; mtk_disp; pro_info; preloader; recovery pro_info; devmap; dkb; gps; gsensor; hdmitx; hwmsensor; kb; logo; misc; misc-sd; nvram; rtc0; sec; seccfg ; stpwmt touch; ttyMT2 ; wmtWifi; wmtdetect	r-
cpuctl	rw-
	r-x

app that performs critical functionalities based on received cell broadcasts. It registers the broadcast receiver `PrivilegedCellBroadcastReceiver` that allows receiving emergency broadcasts from the cell providers (e.g., evacuation alerts, presidential alerts, amber alerts, etc.) and displaying corresponding alerts. This critical functionality can be triggered if the action `android.provider.Telephony.SMS_EMERGENCY_CB_RECEIVED` is received. Our Cross-Vendor and Cross-Version analyses discovered a protection mismatch on this receiver among several devices. For instance, on Nexus S 4G 4.1.1, this receiver is protected with the `Signature` permission `android.permission.BROADCAST_SMS`. However, on other devices (e.g., Nexus6 5.1.1 and MotoX XT1095 5.0.1), it is protected with the `Dangerous` permission `android.permission.READ_PHONE_STATE`. Our investigation reveals that this is also due to the duplicate receivers risky pattern (Section 6.5). On the victim devices, `PrivilegedCellBroadcastReceiver` has been declared twice such that its first declaration requires a `Signature` permission and handles the action `android.provider.Telephony.SMS_EMERGENCY_CB_RECEIVED`, while the second declaration handles less privileged actions and requires a `Dangerous` permission. As discussed, any third-party app can bypass the permission requirement on the first receiver through explicit invocation. We wrote an attack app that was able to trigger this receiver and show various emergency alerts.

Tampering with system wide settings. SystemUI is a preloaded app that controls system windows. It handles and draws a lot of system UIs such as top status bar, system notification and dialogs. To manage the top status bar, the custom Samsung SystemUI includes a service `com.android.systemui.PhoneSettingService`, which handles incoming requests to turn on/off a variety of system wide settings appearing on the top status bar. These settings include turning on/off wifi, bluetooth, location, mobile data, nfc, driving mode, etc; that are usually done with user consent. Our analysis shows a protection mismatch for this service. On S5(4.4.2) and Note8(4.4.2), this service is protected with a signature permission `com.sec.phonesettingservice.permission.PHONE_SETTING`, while on Note 2, 4.4.2, the service is not protected with any permission. We wrote an attack app

that successfully asks the privileged service to turn on all the settings mentioned above without any permission.

Other Randomly Selected Cases. The impact of inconsistent security configurations are significant. In addition to end-to-end attacks we built, we also randomly sampled 40 inconsistencies and manually analyzed what could happen once they were exploited. Note that due to the lack of physical devices, all we could do is just static analysis to infer possible consequences once an exploit succeeds. Such an analysis may not be accurate, but it is still important for understanding the impacts of inconsistent security configurations. The outcomes of our analysis are shown in Table 5. Please note that we could not assess the impact in 5 cases (heavily obfuscated code), while we confirmed that 2 cases have been hardened via runtime checks.

8 Limitations

In this section, we discuss some limitations of our proposed approach.

Components implementation changes. A static change of a component’s security configurations (visibility or permission protection) might not necessarily indicate a security risk all the time. In fact, a developer might *intentionally* decide to export a component or downgrade its permission protection in the following cases: the component’s operations or supplied data are not privileged anymore or the component’s implementation is hardened via runtime checks of the caller’s identity (e.g., `binder.getCallingUid()` or `Context.checkSelfPermission()` APIs). Our solution pinpoints these possibly *unintentional* risky configurations changes and demands further investigation to confirm whether the change was indeed intentional or not.

Components renaming. Our approach would miss detecting inconsistent configurations of components which have been renamed during the customization. In fact, as Android relies heavily on implicit intents for inter-app communication, vendors might rename their components to reflect their organization identity.

9 Related Work

Security risks in Android customization. The extensive Android vendor customization have been proven to be problematic in prior studies. At the Kernel level, ADDICTED [29] finds under-protected Linux device drivers on customized ROMs by comparing them with their counterparts on AOSP images. Our finding on inconsistent GID to permission mappings demonstrates another

Table 5: Impact of Inconsistent Security Configurations

Inconsistent Configuration Category	Impact	Specific Examples
Permission Protection Change	Change System / App Wide Settings	Xiaomi Cloud Settings, Activate SIM
Removed Protected Broadcasts	Trigger Dangerous Operations and events	Trigger data sync, SMS received Airplane mode active, SIM is full
Non-Protected Content Providers	Data Pollution	Write to system logs, Add contacts Change instant messaging configurations
Non-Protected Content Providers	Data Leaks	Read emails, Read contacts Read blocked contact lists
Non-Protected Services	Trigger Dangerous Operations	Access Location, Bind to printing services Kill specific apps, Trigger backup
Non-Protected Activities	Change System wide Settings	Change Telephony settings, Access hidden activities
Non-Protected Receivers	Trigger Dangerous Operations	Send SMS messages, Trigger fake alerts Alter telephony settings , Issue SIM commands

way that can expose critical device drivers. At the framework/ app level, Harehunter [5] reveals the Hanging Attributes References (Hares) vulnerability caused by the under-regulated Android customization. The Hare vulnerability happens when an attribute is used on a device but the party defining it has been removed. A malicious app can then fill the gap to acquire critical capabilities, by simply disguising as the owner of the attribute. Previous works [13, 14, 25] have also highlighted security issues in the permission and components AC in preloaded apps. Gallo et al [13] analyzed five different devices and concluded that serious security issues such as poorer permission control grow sharply with the level of customization. Other prominent work [25] analyzes the pre-installed apps on 10 factory images and reports the presence of known problems such as over-privilege [11], permission re-delegation [12], etc. Our study is fundamentally different from the above work [25] which finds specific known vulnerabilities on a customized image through conducting a reachability analysis from an open entry point to privileged sinks. Instead, we leverage a differential analysis to point out inconsistencies in components’ protection, and consequently detect unintentionally exposed ones. Our analysis further gives insights about possible reasons behind the exposure.

Demystification of Android security configurations. The high flexibility of Android’s security architecture demands a complete understanding of configurable security parameters. Stowaway [11] and PScout [7] lead the way by mapping individual APIs to the required permission. Understanding these parameters provides the necessary domain knowledge in our feature selection. This understanding has inspired other researchers to detect vulnerabilities in apps. The prevalence of misconfigured content providers, activities and services is studied in [30, 8], respectively. These vulnerabilities are due to developers’ exposing critical components or misinterpreting Android’s security protection. Instead of focusing on analyzing an individual app to find if it is vulnerable, our approach learns from the configurations of the same app on other ROMs to deduct if it should be protected or not.

Android vulnerability analysis. Prior research has also uncovered security issues rooted in non-customized AOSP images. PileUp [26] brings to attention the problematic Android upgrading process. Two recent studies examine the crypto misuse in Android apps [9, 16]. Other works evaluate the security risks resulting from design flaws in the push-cloud messaging [18], in the multi-user architecture [24], in Android app uninstal-lation process [28] and in Android’s Clipboard and shar-ing mechanism [10]. Other researchers [20, 15] focused on uncovering vulnerabilities within specific Android apps in the web landscape. These vulnerabili-ties are complementary to the security issues detected in vendor customization, and jointly present a more complete picture of Android ecosystem’s security landscape. To analyze Android vulnerabilities, static and dynamic analysis techniques have been proposed to address the special characteristics of Android platform. CHEX [19], Epicc [21], and FlowDroid [6] apply static analysis to perform vulnerability analysis. Other works [23, 22, 17, 27] employ dynamic analysis to accurately understand app’s behaviors. Both techniques are bene-ficial to our research. Dynamic analysis can help us ex-ploit the likely risky inconsistencies, while static analysis can bring the control/data flow of framework/ app code as another security feature into our differential analysis. We will explore these ideas in future work.

10 Conclusion

In this paper, we make the first attempt to systematically detect security configuration changes introduced by An-droid customization. We list the security features applied at various Android layers and leverage differential analy-sis among a large set of custom ROMs to find out if they are consistent across all of them. By comparing security configura-tions of similar images (from the same vendor, running the same OS version, etc.), we can find critical security changes that might have been unintentionally introduced during the customization. Our analysis shows that indeed, customization parties change several config-

urations that can lead to severe vulnerabilities such as private data exposure and privilege escalation.

11 Acknowledgement

We would like to thank our anonymous reviewers for their insightful comments. This project was supported in part by the NSF grant 1318814.

References

- [1] Android Revolution. <http://goo.gl/MVigfq>.
- [2] Factory Images for Nexus Devices. <https://goo.gl/i0RJnN>.
- [3] Huawei ROMs. <http://goo.gl/dYPTe5>.
- [4] Samsung Updates. <http://goo.gl/RVU84V>.
- [5] AAFER, Y., ZHANG, N., ZHANG, Z., ZHANG, X., CHEN, K., WANG, X., ZHOU, X., DU, W., AND GRACE, M. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS '15.
- [6] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND McDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. PLDI '14.
- [7] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM.
- [8] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM.
- [9] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM.
- [10] FAHL, S., HARBACH, M., OLTROGGE, M., MUDERS, T., AND SMITH, M. Hey, you, get off of my clipboard. In *In proceeding of 17th International Conference on Financial Cryptography and Data Security* (2013).
- [11] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), CCS '11, ACM.
- [12] FELT, A. P., WANG, H. J., MOSCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium* (2011).
- [13] GALLO, R., HONGO, P., DAHAB, R., NAVARRO, L. C., KAWAKAMI, H., GALVÃO, K., JUNQUEIRA, G., AND RIBEIRO, L. Security and system architecture: Comparison of android customizations. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2015).
- [14] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)* (Feb. 2012).
- [15] JIN, X., HU, X., YING, K., DU, W., YIN, H., AND PERI, G. N. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA), CCS '14, ACM.
- [16] KIM, S. H., HAN, D., AND LEE, D. H. Predictability of android openssl's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2013), CCS '13, ACM.
- [17] KLIBER, W., FLYNN, L., BHOSALE, A., JIA, L., AND BAUER, L. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (2014), SOAP '14.
- [18] LI, T., ZHOU, X., XING, L., LEE, Y., NAVEED, M., WANG, X., AND HAN, X. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), CCS '14, ACM.
- [19] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), CCS '12.
- [20] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on webview in the android system. ACSAC '11.
- [21] OCTEAU, D., McDANIEL, P., JHA, S., BARTEL, A., BODDEN, E., KLEIN, J., AND LE TRAON, Y. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security* (2013), SEC'13.
- [22] POEPLAU, S., FRATANTONIO, Y., BIANCHI, A., KRUEGEL, C., AND VIGNA, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. NDSS '14.
- [23] RASTOGI, V., CHEN, Y., AND ENCK, W. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2013), CODASPY '13.
- [24] RATAZZI, P., AAFER, Y., AHLAWAT, A., HAO, H., WANG, Y., AND DU, W. A systematic security evaluation of Android's multi-user framework. In *Mobile Security Technologies (MoST) 2014* (San Jose, CA, USA, 2014), MoST'14.
- [25] WU, L., GRACE, M., ZHOU, Y., WU, C., AND JIANG, X. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer communications security* (New York, NY, USA, 2013), CCS '13, ACM.
- [26] XING, L., PAN, X., WANG, R., YUAN, K., AND WANG, X. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (2014), SP '14.
- [27] YAN, L. K., AND YIN, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), Security'12.
- [28] ZHANG, X., YING, K., AAFER, Y., QIU, Z., AND DU, W. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *NDSS* (2016).
- [29] ZHOU, X., LEE, Y., ZHANG, N., NAVEED, M., AND WANG, X. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA*.
- [30] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in android applications. In *NDSS* (2013).

12 Appendix

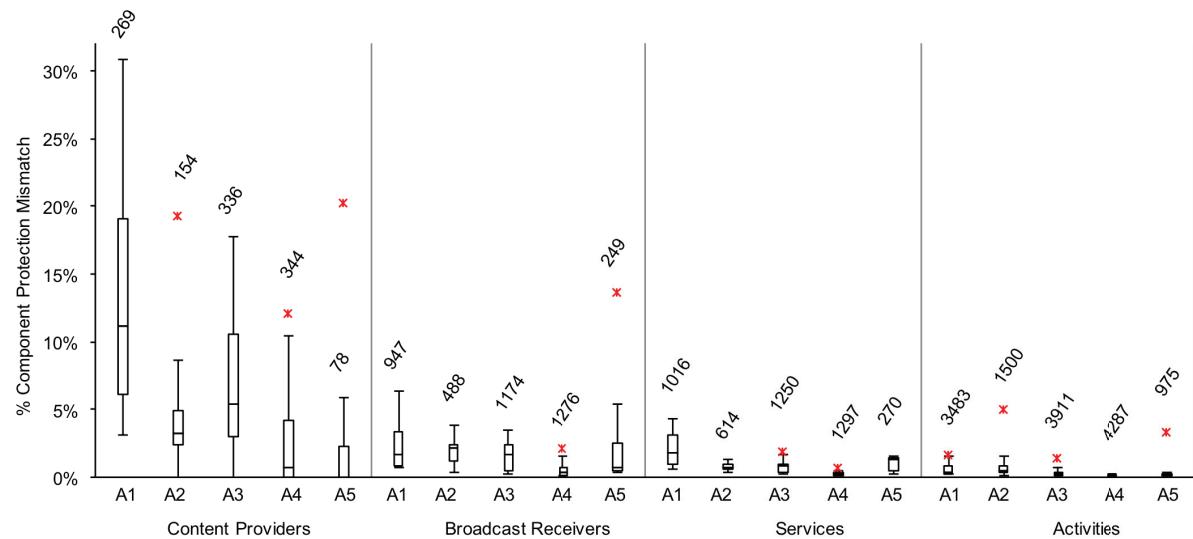


Figure 8: Components Protection Mismatch Breakdown

Identifying and characterizing Sybils in the Tor network

Philipp Winter^{*†}

Roya Ensafi^{*}

Karsten Loesing[‡]

Nick Feamster^{*}

^{*}Princeton University

[†]Karlstad University

[‡]The Tor Project

Abstract

Being a volunteer-run, distributed anonymity network, Tor is vulnerable to Sybil attacks. Little is known about real-world Sybils in the Tor network, and we lack practical tools and methods to expose Sybil attacks. In this work, we develop *sybilhunter*, a system for detecting Sybil relays based on their *appearance*, such as configuration; and *behavior*, such as uptime sequences. We used sybilhunter’s diverse analysis techniques to analyze nine years of archived Tor network data, providing us with new insights into the operation of real-world attackers. Our findings include diverse Sybils, ranging from botnets, to academic research, and relays that hijacked Bitcoin transactions. Our work shows that existing Sybil defenses do not apply to Tor, it delivers insights into real-world attacks, and provides practical tools to uncover and characterize Sybils, making the network safer for its users.

1 Introduction

In a Sybil attack, an attacker controls many virtual identities to obtain disproportionately large influence in a network. These attacks take many shapes, such as sockpuppets hijacking online discourse [34]; the manipulation of BitTorrent’s distributed hash table [35]; and, most relevant to our work, relays in the Tor network that seek to deanonymize users [8]. In addition to coining the term “Sybil,”¹ Douceur showed that practical Sybil defenses are challenging, arguing that Sybil attacks are always possible without a central authority [11]. In this work, we focus on Sybils in Tor—relays that are controlled by a single operator. But what harm can Sybils do?

The effectiveness of many attacks on Tor depends on how large a fraction of the network’s traffic—called the

consensus weight—an attacker can observe. As the attacker’s consensus weight grows, the following attacks become easier.

Exit traffic tampering: When leaving the Tor network, a Tor user’s traffic traverses exit relays, the last hop in a Tor circuit. Controlling exit relays, an attacker can eavesdrop on traffic to collect unencrypted credentials, break into TLS-protected connections, or inject malicious content [37, § 5.2].

Website fingerprinting: Tor’s encryption prevents guard relays (the first hop in a Tor circuit) from learning their user’s online activity. Ignoring the encrypted payload, an attacker can still take advantage of flow information such as packet lengths and timings to infer what websites Tor users are visiting [16].

Bridge address harvesting: Users behind censorship systems use private Tor relays—typically called bridges—as hidden stepping stones into the Tor network. It is important that censors cannot obtain all bridge addresses, which is why The Tor Project rate-limits bridge distribution. However, an attacker can harvest bridge addresses by running a middle relay and looking for incoming connections that do not originate from any of the publicly known guard relays [22, § 3.4].

End-to-end correlation: By running both entry guards and exit relays, an attacker can use timing analysis to link a Tor user’s identity to her activity, e.g., learn that *Alice* is visiting *Facebook*. For this attack to work, an attacker must run at least two Tor relays, or be able to eavesdrop on at least two networks [14].

Configuring a relay to forward more traffic allows an attacker to increase her consensus weight. However, the capacity of a single relay is limited by its link bandwidth and, because of the computational cost of cryptography, by CPU. Ultimately, increasing consensus weight

¹The term is a reference to a book in which the female protagonist, Sybil, suffers from dissociative identity disorder [29].

requires an adversary to add relays to the network; we call these additional relays *Sybils*.

In addition to the above attacks, an adversary needs Sybil relays to manipulate onion services, which are TCP servers whose IP address is hidden by Tor. In the current onion service protocol, six Sybil relays are sufficient to take offline an onion service because of a weakness in the design of the distributed hash table (DHT) that powers onion services [4, § V]. Finally, instead of being a direct means to an end, Sybil relays can be a *side effect* of another issue. In Section 5.1, we provide evidence for what appears to be botnets whose zombies are running Tor relays, perhaps because of a misguided attempt to help the Tor network grow.

Motivated by the lack of practical Sybil detection tools, we design and implement heuristics, leveraging our observations that Sybils (*i*) frequently go online and offline simultaneously, (*ii*) share similarities in their configuration, and (*iii*) may change their identity fingerprint—a relay’s fingerprint is the hash over its public key—frequently, to manipulate Tor’s DHT. Three of our four heuristics are automated and designed to run autonomously while one assists in manual analysis by ranking what relays in the network are the most similar to a given reference relay. Our evaluation suggests that our heuristics differ in their effectiveness; one method detected only a small number of incidents, but some of them no other method could detect. Other heuristics produced a large number of results, and seem well-suited to spot the “low hanging fruit.” We implemented these heuristics in a tool, *sybilhunter*, which we subsequently used to analyze 100 GiB worth of archived network data, consisting of millions of files, and dating back to 2007. Finally, we characterize the Sybil groups we discovered. To sum up, we make the following key contributions:

- We design and implement *sybilhunter*, a tool to analyze past and future Tor network data. While we designed it specifically for the use in Tor, our techniques are general in nature and can easily be applied to other distributed systems such as I2P [31].
- We characterize Sybil groups and publish our findings as datasets to stimulate future research.² We find that Sybils run MitM attacks, DoS attacks, and are used for research projects.

The rest of this paper is structured as follows. We begin by discussing related work in Section 2 and give some background on Tor in Section 3. Section 4 presents the design of our analysis tools, which is then followed by experimental results in Section 5. We discuss our results in Section 6 and conclude the paper in Section 7.

²The datasets are available online at <https://nymity.ch/sybilhunting/>.

2 Related work

In his seminal 2002 paper, Douceur showed that only a *central authority* that verifies new nodes as they join the distributed system is guaranteed to prevent Sybils [11]. This approach conflicts with Tor’s design philosophy that seeks to distribute trust and eliminate central points of control. In addition, a major factor contributing to Tor’s network growth is the low barrier of entry, allowing operators to set up relays both quickly and anonymously. An identity-verifying authority would raise that barrier, alienate privacy-conscious relay operators, and impede Tor’s growth. Barring a central authority, researchers have proposed techniques that leverage a resource that is difficult for an attacker to scale. Two categories of Sybil-resistant schemes turned out to be particularly popular, schemes that build on *social constraints* and schemes that build on *computational constraints*. For a broad overview of alternative Sybil defenses, refer to Levine et al. [19].

Social constraints rely on the assumption that it is difficult for an attacker to form trust relationships with honest users, e.g., befriend many strangers on online social networks. Past work leveraged this assumption in systems such as SybilGuard [39], SybilLimit [38], and SybilInfer [6]. Unfortunately, social graph-based defenses do not work in our setting because there is no existing trust relationship between relay operators.³ Note that we could create such a relationship by, e.g., linking relays to their operator’s social networking account, or by creating a “relay operator web of trust,” but again, we believe that such an effort would alienate relay operators and see limited adoption.

Orthogonal to social constraints, computational resource constraints guarantee that an attacker seeking to operate 100 Sybils needs 100 times the computational resources she would have needed for a single virtual identity. Both Borisov [5] and Li et al. [21] used computational puzzles for that purpose. Computational constraints work well in distributed systems where the cost of joining the network is low. For example, a lightweight client is sufficient to use BitTorrent, allowing even low-end consumer devices to participate. However, this is not the case in Tor because relay operations require constant use of bandwidth and CPU. Unlike in many other distributed systems, it is impossible to run 100 Tor relays while not spending the resources for 100 relays. Computational constraints are inherently tied to running a relay.

In summary, we believe that existing Sybil defenses are ill-suited for application in the Tor network; its distinctive features call for customized solutions that con-

³Relay operators can express in their configuration that their relays are run by the same operator, but this denotes an *intra*-person and not an *inter*-person trust relationship.

sider the nature of Tor relays. There has already been some progress towards that direction; namely, The Tor Project has incorporated a number of both implicit and explicit Sybil defenses that are in place as of June 2016. First, directory authorities—the “gatekeepers” of the Tor network—accept at most two relays per IP address to prevent low-resource Sybil attacks [3, 2]. Similarly, Tor’s path selection algorithm ensures that Tor clients never select two relays in the same /16 network [9]. Second, directory authorities automatically assign flags to relays, indicating their status and quality of service. The Tor Project has recently increased the minimal time until relays obtain the `Stable` flag (seven days) and the `HSDir` flag (96 hours). This change increases the cost of Sybil attacks and gives Tor developers more time to discover and block suspicious relays before they get in a position to run an attack. Finally, the operation of a Tor relay causes recurring costs—most notably bandwidth and electricity—which can further restrain an adversary.

3 Background

We now provide necessary background on the Tor network [10]. Tor consists of several thousand volunteer-run relays that are summarized in the *network consensus* that is voted on and published each hour by nine distributed *directory authorities*. The authorities assign a variety of flags to relays:

- Valid:** The relay is valid, i.e., not known to be broken.
- HSDir:** The relay is an onion service directory, i.e., it participates in the DHT that powers Tor onion services.
- Exit:** The relay is an exit relay.
- BadExit:** The relay is an exit relay, but is either misconfigured or malicious, and should therefore not be used by Tor clients.
- Stable:** Relays are stable if their mean time between failure is at least the median of all relays, or at least seven days.
- Guard:** Guard relays are the rarely-changing first hop for Tor clients.
- Running:** A relay is running if the directory authorities could connect to it in the last 45 minutes.

Tor relays are uniquely identified by their *fingerprint*, a Base32-encoded and truncated SHA-1 hash over their public key. Operators can further assign a *nickname* to their Tor relays, which is a string that identifies a relay (albeit not uniquely) and is easier to remember than its pseudo-random fingerprint. Exit relays have an *exit policy*—a list of IP addresses and ports that the relay allows connections to. Finally, operators that run more than one relay are encouraged to configure their relays to be part

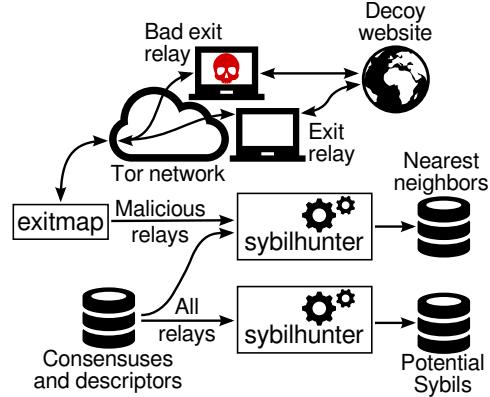


Figure 1: Sybilhunter’s architecture. Two datasets serve as input to sybilhunter; consensuses and server descriptors, and malicious relays gathered with exitmap [37, § 3.1].

of a *relay family*. Families are used to express that a set of relays is controlled by a single operator. Tor clients never use more than one family member in their path to prevent correlation attacks. In February 2016, there were approximately 400 relay families among all 7,000 relays.

4 Data and design

We define Sybils in the Tor network as two or more relays that are controlled by a single person or group of people. Sybils per se do not have to be malicious; a relay operator could simply have forgotten to configure her relays as a relay family. Such Sybils are no threat to the Tor network, which is why we refer to them as *benign Sybils*. What we are interested in is *malicious Sybils* whose purpose is to deanonymize or otherwise harm Tor users.

To uncover malicious Sybils, we draw on two datasets—one publicly available and one created by us. Our detection methods are implemented in a tool, sybilhunter, which takes as input our two datasets and then attempts to expose Sybil groups, as illustrated in Figure 1. Sybilhunter is implemented in Go and consists of 2,300 lines of code.

4.1 Datasets

Figure 1 shows how we use our two datasets. Archived consensuses and router descriptors (in short: descriptors) allow us to (i) restore past states of the Tor network, which sybilhunter mines for Sybil groups, and to (ii) find “partners in crime” of malicious exit relays that we discovered by running exitmap, a scanner for Tor exit relays that we discuss below.

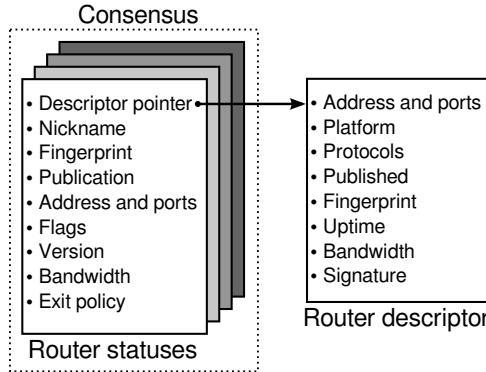


Figure 2: Our primary dataset contains nine years worth of consensuses and router descriptors.

4.1.1 Consensuses and router descriptors

The consensus and descriptor dataset is publicly available on CollecTor [32], an archiving service that is run by The Tor Project. Some of the archived data dates back to 2004, allowing us to restore arbitrary Tor network configurations from the last decade. Not all of CollecTor’s archived data is relevant to our hunt for Sybils, though, which is why we only analyze the following two:

Descriptors Tor relays and bridges periodically upload router descriptors, which capture their configuration, to directory authorities. Figure 2 shows an example in the box to the right. Relays upload their descriptors no later than every 18 hours, or sooner, depending on certain conditions. Note that some information in router descriptors is not verified by directory authorities. Therefore, relays can spoof information such as their operating system, Tor version, and uptime.

Consensuses Each hour, the nine directory authorities vote on their view of all Tor relays that are currently online. The vote produces the consensus, an authoritative list that comprises all running Tor relays, represented as a set of router statuses. Each router status in the consensus contains basic information about Tor relays such as their bandwidth, flags, and exit policy. It also contains a pointer to the relay’s descriptor, as shown in Figure 2. As of June 2016, consensuses contain approximately 7,000 router statuses, i.e., each hour, 7,000 router statuses are published, and archived, by CollecTor.

Table 1 gives an overview of the size of our consensus and descriptor archives. We found it challenging to repeatedly process these millions of files, amounting to more than 100 GiB of uncompressed data, so we implemented a custom parser in Go [36].

Dataset	# of files	Size	Time span
Consensuses	72,061	51 GiB	10/2007–01/2016
Descriptors	34,789,777	52 GiB	12/2005–01/2016

Table 1: An overview of our primary dataset; consensuses and server descriptors since 2007 and 2005, respectively.

4.1.2 Malicious exit relays

In addition to our publicly available and primary dataset, we collected malicious exit relays over 18 months. We call exit relays malicious if they modify forwarded traffic in bad faith, e.g., to run man-in-the-middle attacks. We add these relays to our dataset because they frequently *surface in groups*, as malicious Sybils, because an attacker runs the same attack on several, physically distinct exit relays. Winter et al.’s work [37, § 5.2] further showed that attackers make an effort to stay under the radar, which is why we cannot only rely on active probing to find such relays. We also seek to find potential “partners in crime” of each newly discovered malicious relay, which we discuss in Section 4.3.4.

We exposed malicious exit relays using Winter et al.’s exitmap tool [37, § 3.1]. Exitmap is a Python-based scanning framework for Tor exit relays. Exitmap modules perform a network task that can then be run over all exit relays. One use case is HTTPS man-in-the-middle detection: A module can fetch the certificate of a web server over all exit relays and then compare its fingerprint with the expected, valid fingerprint. Exposed attacks are sometimes difficult to attribute because an attack can take place upstream of the exit relay, e.g., at a malicious autonomous system. However, attribution is only a secondary concern. Our primary concern is protecting Tor users from harm, and we do not need to identify the culprit to do so.

In addition to using the original exitmap modules [37, § 3.1], we implemented modules that detect HTML and HTTP tampering by connecting to a decoy server under our control, and flagging an exit relay as malicious if the returned HTML or HTTP was modified, e.g., to inject data or redirect a user over a transparent HTTP proxy. Since we controlled the decoy server, we knew what our Tor client should get in response. Our modules ran periodically from August 2014 to January 2016, and discovered 251 malicious exit relays whose attacks are discussed in Appendix A. We reported all relays to The Tor Project, which subsequently blocked these relays.

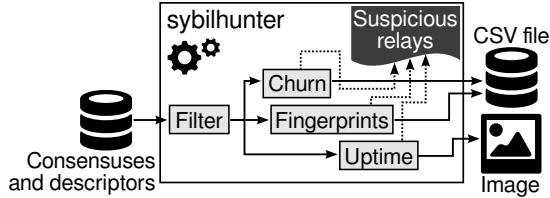


Figure 3: Sybilhunter’s internal architecture. After an optional filtering step, data is then passed on to one of three analysis modules that produce as output either CSV files or an image.

4.2 Threat model

Most of this paper is about applying sybilhunter to archived network data, but we can also apply it to newly incoming data. This puts us in an adversarial setting as attackers can tune their Sybils to evade our system. This is reflected in our adversarial assumptions. We assume that an adversary *does* run more than one Tor relay and exhibits redundancy in their relay configuration, or uptime sequence. An adversary further *can* know how sybilhunter’s modules work, run active or passive attacks, and make a limited effort to stay under the radar, by diversifying parts of their configuration. To detect Sybils, however, our heuristics require *some* redundancy.

4.3 Analysis techniques

Having discussed our datasets and threat model, we now turn to presenting techniques that can expose Sybils. Our techniques are based on the insight that Sybil relays frequently *behave or appear similarly*. Shared configuration parameters such as port numbers and nicknames cause similar appearance whereas Sybils behave similarly when they reboot simultaneously, or exhibit identical quirks when relaying traffic.

Sybilhunter can analyze (*i*) historical network data, dating back to 2007; (*ii*) online data, to detect new Sybils as they join the network; and (*iii*) find relays that might be associated with previously discovered, malicious relays. Figure 3 shows sybilhunter’s internal architecture. Tor network data first passes a filtering component that can be used to inspect a subset of the data, e.g., only relays with a given IP address or nickname. The data is then forwarded to one or more modules that implement an analysis technique. These modules work independently, but share a data structure to find suspicious relays that show up in more than one module. Depending on the analysis technique, sybilhunter’s output is either CSV files or images.

While developing sybilhunter, we had to make many design decisions that we tackled by drawing on the experience we gained by manually analyzing numerous Sybil

groups. We iteratively improved our code and augmented it with new features when we experienced operational shortcomings.

4.3.1 Network churn

The churn rate of a distributed system captures the rate of joining and leaving network participants. In the Tor network, these participants are relays. An unexpectedly high churn rate between two subsequent consensuses means that many relays joined or left, which can reveal Sybils and other network issues because many Sybil operators start and stop their Sybils at the same time, to ease administration—they behave similarly.

The Tor Project is maintaining a Python script [15] that determines the number of previously unobserved relay fingerprints in new consensus documents. If that number is greater than or equal to the static threshold 50, the script sends an e-mail alert. We reimplemented the script in sybilhunter and ran it over all archived consensus documents, dating back to 2007. The script raised 47 alerts in nine years, all of which seemed to be true positives, i.e., they should be of interest to The Tor Project. The script did not raise false positives, presumably because the median number of previously unseen fingerprints in a consensus is only six—significantly below the conservative threshold of 50. Yet, the threshold likely causes false negatives, but we cannot determine the false negative rate because we lack ground truth. In addition, The Tor Project’s script does not consider relays that left the network, does not distinguish between relays with different flags, and does not adapt its threshold as the network grows. We now present an alternative approach that is more flexible and robust.

We found that churn anomalies worthy of our attention range from *flat hills* (Figure 4) to *sudden spikes* (Figure 5). Flat hills can be a sign of an event that affected a large number of relays, over many hours or days. Such an event happened shortly after the Heartbleed bug, when The Tor Project asked relay operators to generate new keys. Relay operators acted gradually, most within two days. Sudden spikes can happen if an attacker adds many relays, all at once. These are mere examples, however; the shape of a time series cannot tell us anything about the nature of the underlying incident.

To quantify the churn rate α between two subsequent consensus documents, we adapt Godfrey et al.’s formula, which yields a churn value that captures both systems that joined and systems that left the network [13, § 2.1]. However, an unusually low number of systems that left could cancel out an unusually high number of new systems and vice versa—an undesired property for a technique that should spot abnormal changes. To address this issue, we split the formula in two parts, creating a

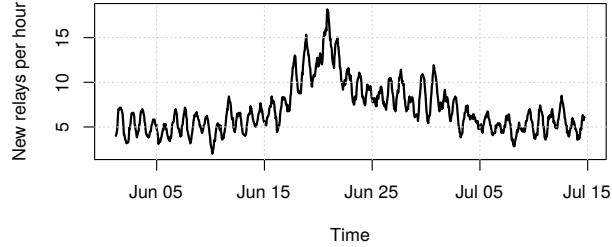


Figure 4: A flat hill of new relays in 2009. The time series was smoothed using a moving average with a window size of 12 hours.

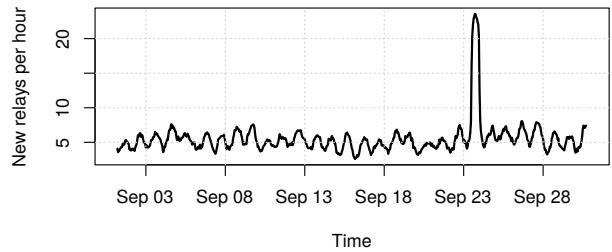


Figure 5: A sudden spike of new relays in 2010. The time series was smoothed using a moving average with a window size of 12 hours.

time series for new relays (α_n) and for relays that left (α_l). C_t is the network consensus at time t , and \ denotes the complement between two consensuses, i.e., the relays that are in the left operand, but not the right operand. We define α_n and α_l as

$$\alpha_n = \frac{|C_t \setminus C_{t-1}|}{|C_t|} \quad \text{and} \quad \alpha_l = \frac{|C_{t-1} \setminus C_t|}{|C_{t-1}|}. \quad (1)$$

Both α_n and α_l are bounded to the interval $[0, 1]$. A churn value of 0 indicates no change between two subsequent consensuses whereas a churn value of 1 indicates a complete turnover. Determining $\alpha_{n,l}$ for the sequence $C_t, C_{t-1}, \dots, C_{t-n}$, yields a time series of churn values that can readily be inspected for abnormal spikes. Figure 6 illustrates the maximum number of Sybils an attacker can add to the network given a threshold for α . The figure shows both the theoretical maximum and a more realistic estimate that accounts for noise, i.e., the median number of new relays in each consensus, which is 73.⁴ We found that many churn anomalies are caused by relays that share a flag, or a flag combination, e.g., HSDir (onion service directories) and Exit (exit relays). Therefore, sybilhunter can also generate per-flag churn time series that can uncover patterns that would be lost in a flag-agnostic time series.

⁴Note that this analysis is “memoryless” and includes relays that have been online before; unlike the analysis above that considered only previously unobserved relays, for which the median number was six.

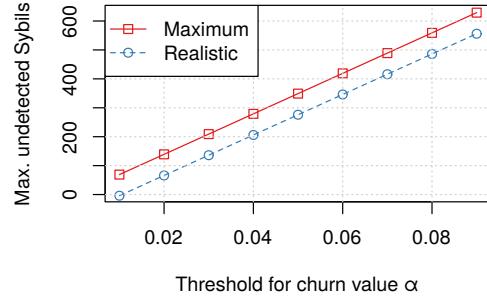


Figure 6: The number of new Sybils (y axis) that can remain undetected given a threshold for the churn value α (x axis). The diagram shows both the maximum and a more realistic estimate that accounts for the median number of new relays in consensuses.

Finally, to detect changes in the underlying time series trend—flat hills—we can smooth $\alpha_{n,l}$ using a simple moving average λ defined as

$$\lambda = \frac{1}{w} \cdot \sum_{i=0}^w \alpha_i. \quad (2)$$

As we increase the window size w , we can detect more subtle changes in the underlying churn trend. If λ or $\alpha_{n,l}$ exceed a manually defined threshold, an alert is raised. Section 5.3 elaborates on how we can select a threshold in practice.

4.3.2 Uptime matrix

For convenience, Sybil operators are likely to administer their relays simultaneously, i.e., update, configure, and reboot them all at the same time. This is reflected in their relays’ uptime. An operating system upgrade that requires a reboot of Sybil relays will induce a set of relays to go offline and return online in a synchronized manner. To isolate such events, we are visualizing the *uptime patterns* of Tor relays by grouping together relays whose uptime is highly correlated. The churn technique presented above is similar but it only provides an aggregate, high-level view on how Tor relays join and leave the network. Since the technique is aggregate, it is poorly suited for visualizing the uptime of specific relays; an abnormally high churn value attracts our attention but does not tell us what caused the anomaly. To fill this gap, we complement the churn analysis with an uptime matrix that we will now present.

This uptime matrix consists of the uptime patterns of all Tor relays, which we represent as binary sequences. Each hour, when a new consensus is published, we add a new data point—“online” or “offline”—to each Tor relay’s sequence. We visualize all sequences in a bitmap whose rows represent consensuses and whose columns

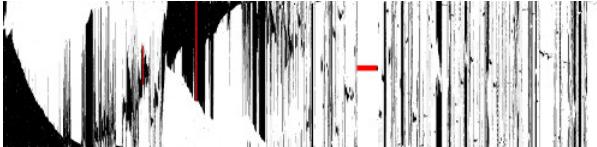


Figure 7: The uptime matrix for 3,000 Tor relays for all of November 2012. Rows represent consensuses and columns represent relays. Black pixels mean that a relay was online, and white means offline. Red blocks denote relays with identical uptime.

represent relays. Each pixel denotes the uptime status of a particular relay at a particular hour. Black pixels mean that the relay was online and white pixels mean that the relay was offline. This type of visualization was first proposed by Ensafi and subsequently implemented by Fifield [12].

Of particular importance is how the uptime sequences are sorted. If highly correlated sequences are not adjacent in the visualization, we might miss them. We sort sequences using single-linkage clustering, a type of hierarchical clustering algorithm that forms groups bottom-up, based on the minimum distance between group members. For our distance function, similar to Andersen et al. [1, § II.B], we use Pearson’s correlation coefficient because it tells us if two uptime sequences change together. The sample correlation coefficient r yields a value in the interval $[-1, 1]$. A coefficient of -1 denotes perfect anti-correlation (relay R_1 is only online when relay R_2 is offline) and 1 denotes perfect correlation (relay R_1 is only online when relay R_2 is online). We define our distance function as $d(r) = 1 - r$, so two perfectly correlated sequences have a distance of zero while two perfectly anti-correlated sequences have a distance of two. Once all sequences are sorted, we color five or more adjacent sequences in red if their uptime sequence is identical. Figure 7 shows an example of our visualization algorithm, the uptime matrix for a subset of all Tor relays in November 2012.

4.3.3 Fingerprint analysis

The information a Tor client needs to connect to an onion service is stored in a DHT that consists of a subset of all Tor relays, the onion service directories (HSDirs). As of June 2016, 47% of all Tor relays serve as HSDirs. A daily-changing set of six HSDirs hosts the contact information of any given onion service. Tor clients contact one of these six HSDirs to request information about the onion service they intend to connect to. A HSDir becomes responsible for an onion service if the difference between its relay fingerprint and the service’s descriptor ID is smaller than that of any other relay. The descrip-

tor ID is derived from the onion service’s public key, a time stamp, and additional information. All HSDirs are public, making it possible to determine at which position in the DHT an onion service will end up at any point in the future. Attackers can exploit the ability to predict the DHT position by repeatedly generating identity keys until their fingerprint is sufficiently close to the targeted onion service’s index, thus becoming its HSDir [4, § V.A].

We detect relays that change their fingerprint frequently by maintaining a lookup table that maps a relay’s IP address to a list of all fingerprints we have seen it use. We sort the lookup table by the relays that changed their fingerprints the most, and output the results. Note that reboots or newly assigned IP addresses are not an issue for this technique—as long as relays do not lose their long-term keys that are stored on their hard drive, their fingerprint stays the same.

4.3.4 Nearest-neighbor ranking

We frequently found ourselves in a situation where exitmap discovered a malicious exit relay and we were left wondering if there were similar, potentially associated relays. Looking for such relays involved tedious manual work, which we soon started to automate. We needed an algorithm for nearest-neighbor ranking that takes as input a “seed” relay and creates as output a list of all relays, ranked by their similarity to the seed relay. We define similarity as shared configuration parameters such as port numbers, IP addresses, exit policies, or bandwidth values. Our algorithm ranks relays by comparing these configuration parameters.

To quantify the similarity between two relays, we use the Levenshtein distance [18], a distance metric that takes as input two strings and determines the minimum number of modifications—insert, delete, and modify—that are necessary to turn string s_2 into s_1 . Our algorithm turns the router statuses and descriptors of two relays into strings and determines their Levenshtein distance. As an example, consider a simple representation consisting of the concatenation of nickname, IP address, and port. To turn string s_2 into s_1 , six operations are necessary; four modifications (green) and two deletions (red):

```
s1: Foo10.0.0.19001
s2: Bar10.0.0.2549001
```

Our algorithm determines the Levenshtein distance between a “seed” relay and all other relays in a consensus. It then ranks the calculated distances in ascending order. For a consensus consisting of 6,525 relays, our algorithm takes approximately 1.5 seconds to finish.⁵ Note

⁵We measured on an Intel Core i7-3520M CPU at 2.9 GHz, a consumer-grade CPU.

that we designed our ranking algorithm to assist in manual analysis. Unlike the other analysis techniques, it does not require a threshold.

5 Evaluation and results

Equipped with sybilhunter, we applied our techniques to nine years of archived Tor network data. We did not set any thresholds, to capture every single churn value, fingerprint, and uptime sequence, resulting in an unfiltered dataset of several megabytes of CSV files and uptime images. We then sorted this dataset in descending order by severity, and began manually analyzing the most significant incidents, e.g., the largest churn values. In Section 5.1, we begin by characterizing Sybil groups we discovered that way. Instead of providing an exhaustive list of all potential Sybils, we focus on our most salient findings—relay groups that were either clearly malicious or distinguished themselves otherwise.⁶ Afterwards, we explore the impact of sybilhunter’s thresholds in Sections 5.2 to 5.6.

Once we discovered a seemingly harmful Sybil group, we reported it to The Tor Project. To defend against Sybil attacks, directory authorities can either remove a relay from the consensus, or take away its *Valid* flag, which means that the relay is still in the consensus, but Tor clients will not consider it for their first or last hop in a circuit. The majority of directory authorities, i.e., five out of nine, must agree on either strategy. This mechanism is meant to distribute the power of removing relays into the hands of a diverse set of people in different jurisdictions.

5.1 Sybil characterization

Table 2 shows the most interesting Sybil groups we identified. The columns show (*i*) what we believe to be the purpose of the Sybils, (*ii*) when the Sybil group was at its peak size, (*iii*) the ID we gave the Sybils, (*iv*) the number of Sybil fingerprints at its peak, (*v*) the analysis techniques that could discover the Sybils, and (*vi*) a short description. The analysis techniques are abbreviated as “N” (Neighbor ranking), “F” (Fingerprint), “C” (Churn), “U” (Uptime), and “E” (exitmap). We now discuss the most insightful incidents in greater detail.

The “rewrite” Sybils These recurring Sybils hijacked Bitcoin transactions by rewriting Bitcoin addresses in relayed HTML. All relays had the *Exit* flag and replaced onion domains found in a web server’s HTTP response

⁶Our datasets and visualizations are available online, and can be inspected for an exhaustive set of potential Sybils. The URL is <https://nymity.ch/sybilhunting/>.

with an impersonation domain, presumably hosted by the attacker. Interestingly, the impersonation domains shared a prefix with the original. For example, the domain `sigaintevyh2rzvw.onion` was replaced with the impersonation domain `sigaintz7qjj3val.onion` whose first seven digits are identical to the original. The attacker could create shared prefixes by repeatedly generating key pairs until the hash over the public key resembled the desired prefix. Onion domains are generated by determining the SHA-1 hash over the public key, truncating it to its 80 most significant bits, and encoding it in Base32. Each Base32 digit of the 16-digit-domain represents five bits. Therefore, to get an n -digit prefix in the onion domain, 2^{5n-1} operations are required on average. For the seven-digit prefix above, this results in $2^{5 \cdot 7 - 1} = 2^{34}$ operations. The author of scallion [30], a tool for generating vanity onion domains, determined that an nVidia Quadro K2000M, a mid-range laptop GPU, is able to generate 90 million hashes per second. On this GPU, a partial collision for a seven-digit prefix can be found in $2^{34} \cdot \frac{1}{90,000,000} \simeq 190$ seconds, i.e., just over three minutes.

We inspected some of the phishing domains and found that the attackers further replaced the original Bitcoin addresses with addresses that are presumably controlled by the attackers, enabling them to hijack Bitcoin transactions. As a result, we believe that the attack was financially motivated.

The “redirect” Sybils These relays all had the *Exit* flag and tampered with HTTP redirects of exit traffic. To protect their users’ login credentials, some Bitcoin sites would redirect users from their HTTP site to the encrypted HTTPS version. This Sybil group tampered with the redirect and directed users to an impersonation site, resembling the original Bitcoin site, probably to steal credentials. We only observed this attack for Bitcoin sites, but cannot rule out that other sites were not attacked.

Interestingly, the Sybils’ descriptors and consensus entries had less in common than other Sybil groups. They used a small set of different ports, Tor versions, bandwidth values, and their nicknames did not exhibit an easily-recognizable pattern. In fact, the only reason why we know that these Sybils belong together is because their attack was identical.

We discovered three Sybil groups that implemented the redirect attack, each of them beginning to surface when the previous one got blocked. The initial group first showed up in May 2014, with only two relays, but slowly grew over time, until it was finally discovered in January 2015. We believe that these Sybils were run by the same attacker because their attack was identical.

It is possible that this Sybil group was run by the same

Purpose	Peak activity	Group ID	Number	Neighbor F Fingerprint Churn Uptime Exitmap	Description
MitM	Jan 2016	rewrite*	42	E	Replaced onion domains with impersonation site.
	Nov 2015	rewrite*	8	E	Replaced onion domains with impersonation site.
	Jun 2015	rewrite*	55	E	Replaced onion domains with impersonation site.
	Apr 2015	rewrite*	71	U,E	Replaced onion domains with impersonation site.
	Mar 2015	redirect†	24	E	Redirected users to impersonated site.
	Feb 2015	redirect†	17	E	Redirected users to impersonated site.
	Jan 2015	redirect†	26	E	Redirected users to impersonated site.
Botnet	Mar 2014	default	—	N	Likely a Windows-powered botnet. The group features wide geographical distribution, which is uncommon for typical Tor relays.
	Oct 2010	trotsky	649	N	The relays were likely part of a botnet. They appeared gradually, and were all running Windows.
Unknown	Jan 2016	cloudvps	61	C,U	Hosted by Dutch hoster XL Internet Services.
	Nov 2015	11BX1371	150	C,U	All relays were in two /24 networks and a single relay had the Exit flag.
	Jul 2015	DenkoNet	58	U	Hosted on Amazon AWS and only present in a single consensus. No relay had the Exit flag.
	Jul 2015	cloudvps	55	C,U	All relays only had the Running and Valid flag. As their name suggests, the relays were hosted by the Dutch hoster “CloudVPS.”
	Dec 2014	Anonpoke	284	C,U	The relays did not have the Exit flag and were removed from the network before they could get the HSDir flag.
	Dec 2014	FuslVZTOR	246	C,U	The relays showed up only hours after the LizardNSA incident.
	Dec 2014	LizardNSA	4,615	C,U	A group publicly claimed to be responsible for the attack [24]. All relays were hosted in the Google cloud and The Tor Project removed them within hours.
Research	May 2015	fingerprints	168	F	All twelve IP addresses, located in the same /24, changed their fingerprint regularly, presumably in an attempt to manipulate the distributed hash table.
	Mar 2014	FDCservers	264	C,U	Relays that were involved in an experimental onion service deanonymization attack [8].
	Feb 2013	AmazonEC2	1,424	F,C,U	We observed 1,424 relay fingerprints on 88 IP addresses. These Sybils were likely part of a research project [4, § V].
	Jun 2010	planetlab	595	C,U	According to a report from The Tor Project [20], a researcher started these relays to learn more about scalability effects.

Table 2: The most salient Sybil groups that sybilhunter and our exitmap modules discovered. We believe that groups marked with the symbols * and † were run by the same operator, respectively. Note that sybilhunter was unable to detect six Sybil groups in the category “MitM.”

attackers that controlled the “rewrite” group but we have no evidence to support that hypothesis. Interestingly, only our exitmap module was able to spot these Sybils. The relays joined the network gradually over time and had little in common in their configuration, which is why our heuristics failed. In fact, we cannot rule out that the

adversary was upstream of the exit relay, or gained control over these relays.

The “FDCservers” Sybils Attackers used these Sybils to deanonymize onion service users, as discussed by The Tor Project in a July 2014 blog post [8]. Sup-

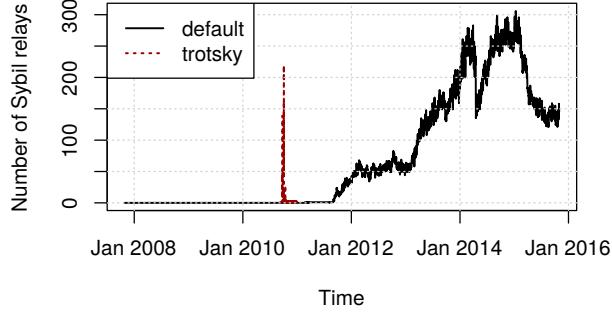


Figure 8: The number of “default” and “trotsky” Sybil members over time.

posedly, CMU/SEI-affiliated researchers were executing a traffic confirmation attack by sending sequences of RELAY_EARLY and RELAY cells as a signal down the circuit to the client, which the reference implementation never does [8, 7]. The attacking relays were both onion service directories and guards, allowing them to control both ends of the circuit for some Tor clients that were fetching onion service descriptors. Therefore, the relays could tell for a fraction of Tor users what onion service they were intending to visit. Most relays were running FreeBSD, used Tor in version 0.2.4.18-rc, had identical flags, mostly identical bandwidth values, and were located in 50.7.0.0/16 and 204.45.0.0/16. All of these shared configuration options made the relays easy to identify.

The relays were added to the network in batches, presumably starting in October 2013. On January 30, 2014, the attackers added 58 relays to the 63 existing ones, giving them control over 121 relays. On July 8, 2014, The Tor Project blocked all 123 IP addresses that were running at the time.

The “default” Sybils This group, named after the Sybils’ shared nickname “default,” has been around since September 2011 and consists of Windows-powered relays only. We extracted relays by filtering consensuses for the nickname “default,” onion routing port 443, and directory port 9030. The group features high IP address churn. For October 2015, we found “default” relays in 73 countries, with the top three countries being Germany (50%), Russia (8%), and Austria (7%). The majority of these relays had little uptime and exhibited a diurnal pattern, suggesting that they were powered off regularly—as it often is the case for desktop computers and laptops.

To get a better understanding of the number of “default” relays over time, we analyzed all consensuses, extracting the number of relays whose nickname was “default,” whose onion routing port was 443, and whose directory port was 9030. We did this for the first consensus

every day and plot the result in Figure 8. Note that we might overestimate the numbers as our filter could capture unrelated relays.

The above suggests that some of the “default” relays are running without the owner’s knowledge. While the relays do not fit the pattern of Sefnit (a.k.a. Mevade) [26] and Skynet [27]—two pieces of malware that use an onion service as command and control server—we believe that the “default” relays constitute a botnet.

The “trotsky” Sybils Similar to the “default” group, the “trotsky” relays appear to be part of a botnet. Most of the relays’ IP addresses were located in Eastern Europe, in particular in Slovenia, Croatia, and Bosnia and Herzegovina. The relays were all running on Windows, in version 0.2.1.26, and listening on port 443. Most of the relays were configured as exits, and The Tor Project assigned some of them the BadExit flag.

The first “trotsky” members appeared in September 2010. Over time, there were two relay peaks, reaching 139 (September 23) and 219 (October 3) relays, as illustrated in Figure 8. After that, only 1–3 relays remained in the consensus.

The “Amazon EC2” Sybils The relays all used randomly-generated nicknames, consisting of sixteen or seventeen letters and numbers; Tor in version 0.2.2.37; GNU/Linux; and IP addresses in Amazon’s EC2 net-block. Each of the 88 IP addresses changed its fingerprint 24 times, but not randomly: the fingerprints were chosen systematically, in a small range. For example, relay 54.242.248.129 had fingerprints with the prefixes 8D, 8E, 8F, and 90. The relays were online for 48 hours. After 24 hours, most of the relays obtained the HSDir flag. This behavior appears to be a clear attempt to manipulate Tor’s DHT.

We believe that this Sybil group was run by Biryukov, Pustogarov, and Weinmann as part of their Security and Privacy 2013 paper “Trawling for Tor Hidden Services” [4]—one of the few Sybil groups that were likely run by academic researchers.

The “Anonpoke” Sybils All relays shared the nickname “Anonpoke” and were online for four hours until they were rejected. All relays were hosted by a VPS provider in the U.S., Rackspace, with the curious exception of a single relay that was hosted in the UK, and running a different Tor version. The relays advertised the default bandwidth of 1 GiB/s on port 9001 and 9030. All relays were middle relays and running as directory mirror. All Sybils were configured to be an onion service directory, but did not manage to get the flag in time.

The “PlanetLab” Sybils A set of relays that used a variation of the strings “planet”, “plab”, “pl”, and “planetlab” as their nickname. The relays’ exit policy allowed ports 6660–6667, but they did not get the Exit flag. The Sybils were online for three days and then removed by The Tor Project, as mentioned in a blog post [20]. The blog post further says that the relays were run by a researcher to learn more about “cloud computing and scaling effects.”

The “LizardNSA” Sybils All relays were hosted in the Google Cloud and only online for ten hours, until the directory authorities started to reject them. The majority of machines were middle relays (96%), but the attackers also started some exit relays (4%). The Sybils were set up to be onion service directories, but the relays were taken offline before they could earn the HSDir flag. If all relays would have obtained the HSDir flag, they would have constituted almost 50% of all onion service directories; the median number of onion service directories on December 26 was 3,551.

Shortly after the attack began, somebody claimed responsibility on the tor-talk mailing list [24]. Judging by the supposed attacker’s demeanor, the attack was mere mischief.

The “FusIVZTOR” Sybils All machines were middle relays and hosted in the netblock 212.38.181.0/24, owned by a UK VPS provider. The directory authorities started rejecting the relays five hours after they joined the network. The relays advertized the default bandwidth of 1 GiB/s and used randomly determined ports. The Sybils were active in parallel to the “LizardNSA” attack, but there is no reason to believe that both incidents were related.

5.2 Alerts per method

Having investigated the different types of alerts our methods raised, we now provide intuition on how many of these alerts we would face in practice. To this end, we first determined conservative thresholds, chosen to yield a manageable number of alerts per week. For network churn, we set the threshold for α_n for relays with the Valid flag to 0.017. For the fingerprint method, we raised an alert if a relay changed its fingerprint at least ten times per month, and for uptime visualizations we raised an alert if at least five relays exhibited an identical uptime sequence. We used a variety of analysis windows to achieve representative results. For example, the Tor network’s churn rate slowly reduced over the years, which is why we only analyzed 2015 and 2016. Table 3 shows the results. For comparison, the table also shows our eximap modules, which did not require any thresholds.

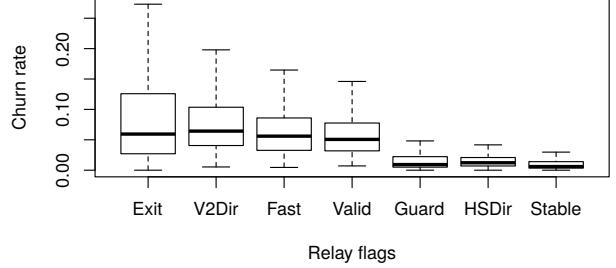


Figure 9: The churn distribution for seven relay flags. We removed values greater than the plot whiskers.

5.3 Churn rate analysis

We determined the churn rate between two subsequent consensuses for all 72,061 consensuses that were published between October 2007 and January 2016. Considering that (i) there are 162 gaps in the archived data, that (ii) we created time series for joining and leaving relays, and that (iii) we determined churn values for all twelve relay flags, we ended up with $(72,061 - 162) \cdot 2 \cdot 12 = 1,725,576$ churn values. Figure 9 shows a box plot for the churn distribution (joining and leaving churn values concatenated) for the seven most relevant relay flags. We removed values greater than the plot whiskers (which extend to values 1.5 times the interquartile range from the box) to better visualize the width of the distributions. Unsurprisingly, relays with the Guard, HSDir, and Stable flag experience the least churn, probably because relays are only awarded these flags if they are particularly stable. Exit relays have the most churn, which is surprising given that exit relays are particularly sensitive to operate. Interestingly, the median churn rate of the network has steadily decreased over the years, from 0.04 in 2008 to 0.02 in 2015.

Figure 10 illustrates churn rates for five days in August 2008, featuring the most significant anomaly in our data. On August 19, 822 relays left the network, resulting in a sudden spike, and a baseline shift. The spike was caused by the Tor network’s switch from consensus format version three to four. The changelog says that in version four, routers that do not have the Running flag are no longer listed in the consensus.

To alleviate the choice of a detection threshold, we plot the number of alerts (in log scale) in 2015 as the threshold increases. We calculate these numbers for three simple moving average window sizes. The result is shown in Figure 11. Depending on the window size, thresholds greater than 0.012 seem practical considering that 181 alerts per year average to approximately one alert in two days—a tolerable number of incidents to investigate. Unfortunately, we are unable to determine the false positive rate because we do not have ground truth.

Method	Analysis window	Threshold	Total alerts	Alerts per week
Fingerprint	10/2007–01/2016	10	551	1.3
Churn	01/2015–01/2016	0.017	110	1.9
Uptimes	01/2009–01/2016	5	3,052	8.3
Exitmap	08/2014–01/2016	—	251	3.2

Table 3: The number of alerts our methods raised. We used different analysis windows for representative results, and chose conservative thresholds to keep the number of alerts per week manageable.

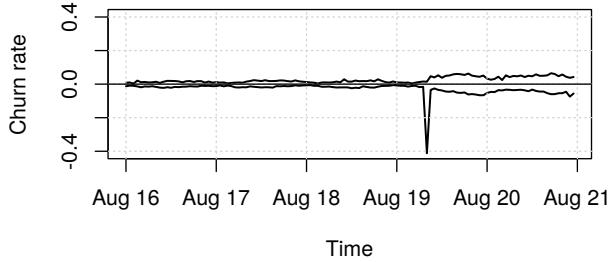


Figure 10: In August 2008, an upgrade in Tor’s consensus format caused the biggest anomaly in our dataset. The positive time series represents relays that joined and the negative one represents relays that left.

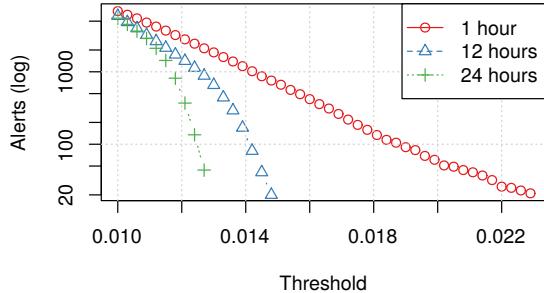


Figure 11: The number of alerts (in log scale) in 2015 as the detection threshold increases, for three smoothing window sizes.

5.4 Uptime analysis

We generated relay uptime visualizations for each month since 2007, resulting in 100 images. We now discuss a subset of these images, those containing particularly interesting patterns.

Figure 12 shows June 2010, featuring a clear “Sybil block” in the center. The Sybils belonged to a researcher who, as documented by The Tor Project [20], started several hundred Tor relays on PlanetLab for research on scalability (the “PlanetLab” Sybils discussed above). Our manual analysis could verify this. The relays were easy to identify because their nicknames suggested that they were hosted on PlanetLab, containing strings such as “planetlab,” “planet,” and “plab.” Note the small



Figure 12: In June 2010, a researcher started several hundred Tor relays on PlanetLab [20]. The image shows the uptime of 2,000 relays for all of June.

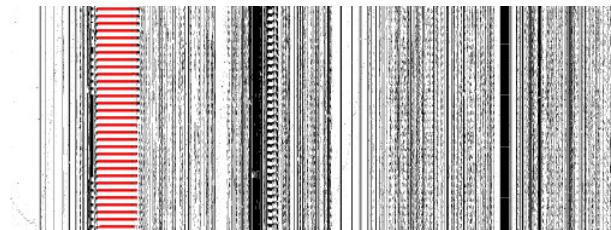


Figure 13: August 2012 featured a curious “step pattern,” caused by approximately 100 Sybils. The image shows the uptime of 2,000 relays for all of August.

height of the Sybil block, indicating that the relays were only online for a short time.

Figure 13 features a curious “step pattern” for approximately 100 relays, all of which were located in Russia and Germany. The relays appeared in December 2011, and started exhibiting the diurnal step pattern (nine hours uptime followed by fifteen hours downtime) in March 2012. All relays had similar nicknames, consisting of eight seemingly randomly-generated characters. In April 2013, the relays finally disappeared.

Figure 14 illustrates the largest Sybil group to date, comprising 4,615 Tor relays (the “LizardNSA” Sybils discussed above). An attacker set up these relays in the Google cloud in December 2014. Because of its magnitude, the attack was spotted almost instantly, and The Tor Project removed the offending relays only ten hours after they appeared.



Figure 14: In December 2014, an attacker started several thousand Tor relays in the Google cloud. The image shows the uptime of 4,000 relays for all of December.

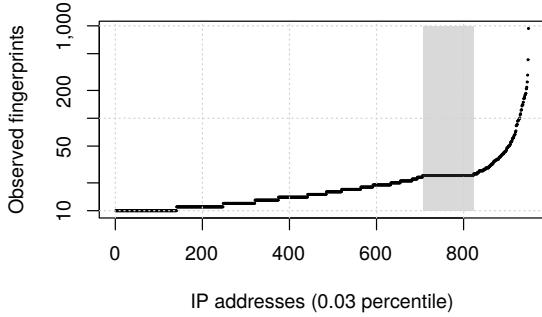


Figure 15: The number of observed fingerprints for the 1,000 relays that changed their fingerprints the most.

5.5 Fingerprint anomalies

We determined how often all Tor relays changed their fingerprint from 2007 to 2015. Figure 15 illustrates the number of fingerprints (y axis) we have observed for the 1,000 Tor relays (x axis) that changed their fingerprint the most. All these relays changed their fingerprint at least ten times. Twenty-one relays changed their fingerprint more than 100 times, and the relay at the very right end of the distribution changed its fingerprint 936 times. This relay’s nickname was “openwrt,” suggesting that it was a home router that was rebooted regularly, presumably losing its long-term keys in the process. The relay was running from August 2010 to December 2010.

Figure 15 further contains a peculiar plateau, shown in the shaded area between index 707 and 803. This plateau was caused by a group of Sybils, hosted in Amazon EC2, that changed their fingerprint exactly 24 times (the “Amazon EC2” Sybils discussed above). Upon inspection, we noticed that this was likely an experiment for a Security and Privacy 2013 paper on deanonymizing Tor onion services [4, § V].

We also found that many IP addresses in the netblock 199.254.238.0/24 frequently changed their fingerprint. We contacted the owner of the address block and were told that the block used to host VPN services. Apparently, several people started Tor relays and since the VPN service would not assign permanent IP addresses, the Tor relays would periodically change their address, causing the churn we observe.

5.6 Accuracy of nearest-neighbor ranking

Given a Sybil relay, how good is our nearest-neighbor ranking at finding the remaining Sybils? To answer this question, we now evaluate our algorithm’s accuracy, which we define as the fraction of neighbors it correctly labels as Sybils. For example, if eight out of ten Sybils are correctly labeled as neighbors, the accuracy is 0.8.

A sound evaluation requires ground truth, i.e., relays that are *known* to be Sybils. All we have, however, are relays that we *believe* to be Sybils. In addition, the number of Sybils we found is only a lower bound—we are unlikely to have detected all Sybil groups. Therefore, our evaluation is doomed to overestimate our algorithm’s accuracy because we are unable to test it on the Sybils we did not discover.

We evaluate our ranking algorithm on two datasets; the “bad exit” Sybil groups from Table 5, and relay families. We chose the bad exit Sybils because we observed them running identical, active attacks, which makes us confident that they are in fact Sybils. Recall that a relay family is a set of Tor relays that is controlled by a single operator, but configured to express this mutual relationship in the family members’ configuration file. Therefore, relay families are benign Sybils. As of January 2016, approximately 400 families populate the Tor network, ranging in size from only two to 25 relays.

We evaluate our algorithm by finding the nearest neighbors of a family member. Ideally, all neighbors are family members, but the use of relay families as ground truth is very likely to overestimate results because family operators frequently configure their relays identically on purpose. At the time of this writing, a popular relay family has the nicknames “AccessNow000” to “AccessNow009,” adjacent IP addresses, and identical contact information—perfect prerequisites for our algorithm. We expect the operators of *malicious* Sybils, however, to go out of their way to obscure the relationship between their relays.

To determine our algorithm’s accuracy, we used all relay families that were present in the first consensus that was published in October 2015. For each relay that had at least one mutual family relationship, we determined its $n - 1$ nearest neighbors where n is the family size. Basically, we evaluated how good our algorithm is at finding the relatives of a family member. We determined the accuracy—a value in $[0, 1]$ —for each family member. The result is shown in Figure 16(b), a distribution of accuracy values.

Next, we repeated the evaluation with the bad exit Sybil groups from Table 5. Again, we determined the $n - 1$ nearest neighbors of all bad exit relays, where n is the size of the Sybil group. The accuracy is the fraction of relays that our algorithm correctly classified as neigh-

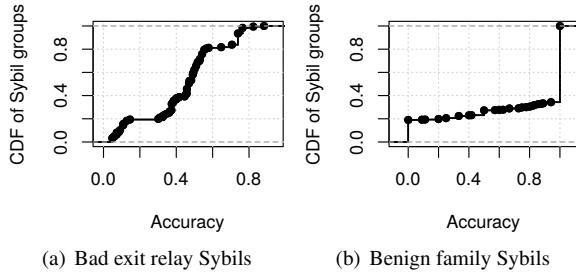


Figure 16: ECDF for our two evaluations, the bad exit Sybils in Fig. 16(a) and the benign family Sybils in Fig. 16(b).

Method	Analysis window	Run time
Churn	Two consensuses	~0.2s
Neighbor ranking	One consensus	~1.6s
Fingerprint	One month	~58.0s
Uptimes	One month	~145.0s

Table 4: The computational cost of our analysis techniques.

bor. The result is illustrated in Figure 16(a).

As expected, our algorithm is significantly more accurate for the family dataset—66% of rankings had perfect accuracy. The bad exit dataset, however, did worse. Not a single ranking had perfect accuracy and 59% of all rankings had an accuracy in the interval [0.3, 0.6]. Nevertheless, we find that our algorithm facilitates manual analysis given how quickly it can provide us with a list of the most similar relays. Besides, inaccurate results (i.e., similar neighbors that are not Sybils) are cheap as sybilhunter users would not spend much time on neighbors that bear little resemblance to the “seed” relay.

5.7 Computational cost

Fast techniques lend themselves to being run hourly, for every new consensus, while slower ones must be run less frequent. Table 4 gives an overview of the runtime of our methods.⁷ We stored our datasets on a solid state drive to eliminate I/O as performance bottleneck.

The table columns contain, from left to right, our analysis technique, the technique’s analysis window, and how long it takes to compute its output. Network churn calculation is very fast; it takes as input only two consensus files and can easily be run for every new network consensus. Nearest-neighbor ranking takes approximately 1.6 seconds for a single consensus counting 6,942 relays. Fingerprint and uptime analysis for one month worth of

⁷We determined all performance numbers on an Intel Core i7-3520M CPU at 2.9 GHz, a consumer-grade CPU.

consensuses takes approximately one and two minutes, respectively—easy to invoke daily, or even several times a day.

6 Discussion

Having used sybilhunter in practice for several months, we now elaborate on both our operational experience and the shortcomings we encountered.

6.1 Operational experience

Our practical work with sybilhunter taught us that analyzing Sybils frequently requires manual verification, e.g., (i) comparing an emerging Sybil group with a previously disclosed one, (ii) using exitmap to send decoy traffic over Sybils, or (iii) sorting and comparing information in relay descriptors. We found that the amount of manual work greatly depends on the Sybils under investigation. The MitM groups in Table 2 were straightforward to spot—in a matter of minutes—while the botnets required a few hours of effort. It is difficult to predict all analysis scenarios that might arise in the future, so we designed sybilhunter to be interoperable with Unix command line tools [28]. Sybilhunter’s CSV-formatted output can easily be piped into tools such as sed, awk, and grep. We found that compact text output was significantly easier to process, both for plotting and for manual analysis. Aside from Sybil detection, sybilhunter can serve as valuable tool to better understand the Tor network and monitor its reliability. Our techniques have disclosed network consensus issues and can illustrate the diversity of Tor relays, providing empirical data that can support future network design decisions.

A key issue in the arms race of eliminating harmful relays lies in *information asymmetry*. Our detection techniques and code are freely available while our adversaries operate behind closed doors, creating an uphill battle that is difficult to sustain given our limited resources. In practice, we can reduce this asymmetry and limit our adversaries’ knowledge by keeping secret sybilhunter’s thresholds and exitmap’s detection modules, so our adversary is left guessing what our tools seek to detect. This differentiation between an *open* analysis framework such as the one we discuss in this paper, and *secret* configuration parameters seems to be a sustainable trade-off. Note that we are not arguing in favor of the flawed practice of security by obscurity. Instead, we are proposing to add a layer of obscurity *on top* of existing defense layers.

We are working with The Tor Project on incorporating our techniques in Tor Metrics [33], a website containing network visualizations that are frequented by numerous volunteers. Many of these volunteers discover anomalies and report them to The Tor Project. By incorporating

our techniques, we hope to benefit from “crowd-sourced” Sybil detection.

6.2 Limitations

In Section 4.2, we argued that we are unable to expose all Sybil attacks, so our results represent a lower bound. An adversary unconstrained by time and money can add an unlimited number of Sybils to the network. Indeed, Table 2 contains six Sybil groups that sybilhunter was unable to detect. Fortunately, exitmap was able to expose these Sybils, which emphasizes the importance of diverse and complementary analysis techniques. Needless to say, sybilhunter works best when analyzing attacks that took place before we built sybilhunter. Adversaries that know of our methods can evade them at the cost of having to spend time and resources. To evade our churn and uptime heuristics, Sybils must be added and modified independently over time. Evasion of our fingerprint heuristic, e.g., to manipulate Tor’s DHT, requires more physical machines. Finally, manipulation of our neighbor ranking requires changes in configuration. This arms race is unlikely to end, barring fundamental changes in how Tor relays are operated.

Sybilhunter is unable to ascertain the purpose of a Sybil attack. While the purpose is frequently obvious, Table 2 contains several Sybil groups that we could not classify. In such cases, it is difficult for The Tor Project to make a call and decide if Sybils should be removed from the network. Keeping them runs the risk of exposing users to an unknown attack, but removing them deprives the network of bandwidth. Often, additional context is helpful in making a call. For example, Sybils that are (i) operated in “bulletproof” autonomous systems [17, § 2], (ii) show signs of not running the Tor reference implementation, or (iii) spoof information in their router descriptor all suggest malicious intent. In the end, Sybil groups have to be evaluated case by case, and the advantages and disadvantages of blocking them have to be considered.

Finally, there is significant room for improving our nearest neighbor ranking. For simplicity, our algorithm represents relays as strings, ignoring a wealth of nuances such as topological proximity of IP addresses, or predictable patterns in port numbers.

7 Conclusion

We presented sybilhunter, a novel system that uses diverse analysis techniques to expose Sybils in the Tor network. Equipped with this tool, we set out to analyze nine years of The Tor Project’s archived network data. We discovered numerous Sybil groups, twenty of which we present in this work. By analyzing the Sybil

groups sybilhunter discovered, we found that (i) Sybil relays are frequently configured very similarly, and join and leave the network simultaneously; (ii) attackers differ greatly in their technical sophistication; and (iii) our techniques are not only useful for spotting Sybils, but turn out to be a handy analytics tool to monitor and better understand the Tor network. Given the lack of a central identity-verifying authority, it is always possible for well-executed Sybil attacks to stay under our radar, but we found that a complementary set of techniques can go a long way towards finding malicious Sybils, making the Tor network more secure and trustworthy for its users.

All our code, data, visualizations, and an open access bibliography of our references are available online at <https://nymity.ch/sybilhunting/>.

Acknowledgments

We want to thank our shepherd, Tudor Dumitraş, for his guidance on improving our work. We also want to thank Georg Koppen, Prateek Mittal, Stefan Lindskog, the Tor developers, and the wider Tor community for helpful feedback. This research was supported in part by the Center for Information Technology Policy at Princeton University and by the National Science Foundation Awards CNS-1540055 and CNS-1602399.

References

- [1] David G. Andersen et al. “Topology Inference from BGP Routing Dynamics”. In: *Internet Measurement Workshop*. ACM, 2002. URL: <https://nymity.ch/sybilhunting/pdf/Andersen2002a.pdf> (cit. on p. 7).
- [2] Kevin Bauer and Damon McCoy. *No more than one server per IP address*. Mar. 2007. URL: <https://gitweb.torproject.org/torspec.git/tree/proposals/109-no-sharing-ips.txt> (cit. on p. 3).
- [3] Kevin Bauer et al. “Low-Resource Routing Attacks Against Tor”. In: *WPES*. ACM, 2007. URL: <https://nymity.ch/sybilhunting/pdf/Bauer2007a.pdf> (cit. on p. 3).
- [4] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. “Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization”. In: *Security & Privacy*. IEEE, 2013. URL: <https://nymity.ch/sybilhunting/pdf/Biryukov2013a.pdf> (cit. on pp. 2, 7, 9, 10, 13).
- [5] Nikita Borisov. “Computational Puzzles as Sybil Defenses”. In: *Peer-to-Peer Computing*. IEEE, 2005. URL: <https://nymity.ch/sybilhunting/pdf/Borisov2006a.pdf> (cit. on p. 2).
- [6] George Danezis and Prateek Mittal. “SybilInfer: Detecting Sybil Nodes using Social Networks”. In: *NDSS*. The Internet Society, 2009. URL: <https://nymity.ch/sybilhunting/pdf/Danezis2009a.pdf> (cit. on p. 2).
- [7] Roger Dingledine. *Did the FBI Pay a University to Attack Tor Users?* Nov. 2015. URL: <https://blog.torproject.org/blog/did-fbi-pay-university-attack-tor-users> (cit. on p. 10).
- [8] Roger Dingledine. *Tor security advisory: “relay early” traffic confirmation attack*. July 2014. URL: <https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack> (cit. on pp. 1, 9, 10).

- [9] Roger Dingledine and Nick Mathewson. *Tor Path Specification*. URL: <https://gitweb.torproject.org/torspec.git/tree/path-spec.txt> (cit. on p. 3).
- [10] Roger Dingledine, Nick Mathewson, and Paul Syverson. “Tor: The Second-Generation Onion Router”. In: *USENIX Security*. USENIX, 2004. URL: <https://nymity.ch/sybilhunting/pdf/Dingledine2004a.pdf> (cit. on p. 3).
- [11] John R. Douceur. “The Sybil Attack”. In: *Peer-to-Peer Systems*. 2002. URL: <https://nymity.ch/sybilhunting/pdf/Douceur2002a.pdf> (cit. on pp. 1, 2).
- [12] David Fifield. #12813—Look at a bitmap visualization of relay consensus. 2014. URL: <https://bugs.torproject.org/12813> (cit. on p. 7).
- [13] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. “Minimizing Churn in Distributed Systems”. In: *SIGCOMM*. ACM, 2006. URL: <https://nymity.ch/sybilhunting/pdf/Godfrey2006a.pdf> (cit. on p. 5).
- [14] Aaron Johnson et al. “Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries”. In: *CCS*. ACM, 2013. URL: <https://nymity.ch/sybilhunting/pdf/Johnson2013a.pdf> (cit. on p. 1).
- [15] Damian Johnson. *doctor – service that periodically checks the Tor network for consensus conflicts and other hiccups*. URL: <https://gitweb.torproject.org/doctor.git/tree/> (cit. on p. 5).
- [16] Marc Juarez et al. “A Critical Evaluation of Website Fingerprinting Attacks”. In: *CCS*. ACM, 2014. URL: <https://nymity.ch/sybilhunting/pdf/Juarez2014a.pdf> (cit. on p. 1).
- [17] Maria Konte, Roberto Perdisci, and Nick Feamster. “ASwatch: An AS Reputation System to Expose Bulletproof Hosting ASes”. In: *SIGCOMM*. ACM, 2015. URL: <https://nymity.ch/sybilhunting/pdf/Konte2015a.pdf> (cit. on p. 15).
- [18] Vladimir Iosifovich Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals”. In: *Soviet Physics-Doklady* 10.8 (1966). URL: <https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf> (cit. on p. 7).
- [19] Brian Neil Levine, Clay Shields, and N. Boris Margolin. *A Survey of Solutions to the Sybil Attack*. Tech. rep. University of Massachusetts Amherst, 2006. URL: <https://nymity.ch/sybilhunting/pdf/Levine2006a.pdf> (cit. on p. 2).
- [20] Andrew Lewman. *June 2010 Progress Report*. June 2010. URL: <https://blog.torproject.org/blog/june-2010-progress-report> (cit. on pp. 9, 11, 12).
- [21] Frank Li et al. “SybilControl: Practical Sybil Defense with Computational Puzzles”. In: *Scalable Trusted Computing*. ACM, 2012. URL: <https://nymity.ch/sybilhunting/pdf/Li2012a.pdf> (cit. on p. 2).
- [22] Zhen Ling et al. “Tor Bridge Discovery: Extensive Analysis and Large-scale Empirical Evaluation”. In: *IEEE Transactions on Parallel and Distributed Systems* 26.7 (2015). URL: <https://nymity.ch/sybilhunting/pdf/Ling2015b.pdf> (cit. on p. 1).
- [23] Zhen Ling et al. “TorWard: Discovery, Blocking, and Traceback of Malicious Traffic Over Tor”. In: *IEEE Transactions on Information Forensics and Security* 10.12 (2015). URL: <https://nymity.ch/sybilhunting/pdf/Ling2015a.pdf> (cit. on p. 17).
- [24] Lizards. Dec. 2014. URL: <https://lists.torproject.org/pipermail/tor-talk/2014-December/036197.html> (cit. on pp. 9, 11).
- [25] Moxie Marlinspike. *sslstrip*. URL: <https://moxie.org/software/sslstrip/> (cit. on p. 17).
- [26] msft-mmpc. *Tackling the Sefnit botnet Tor hazard*. Jan. 2014. URL: <https://blogs.technet.microsoft.com/mmpc/2014/01/09/tackling-the-sefnit-botnet-tor-hazard/> (cit. on p. 10).
- [27] nex. *Skynet, a Tor-powered botnet straight from Reddit*. Dec. 2012. URL: <https://community.rapid7.com/community/infosec/blog/2012/12/06/skynet-a-tor-powered-botnet-straight-from-reddit> (cit. on p. 10).
- [28] Rob Pike and Brian W. Kernighan. “Program Design in the UNIX System Environment”. In: *Bell Labs Technical Journal* 63.8 (1983). URL: <https://nymity.ch/sybilhunting/pdf/Pike1983a.pdf> (cit. on p. 14).
- [29] Flora Rheta Schreiber. *Sybil: The true story of a woman possessed by 16 separate personalities*. Henry Regnery, 1973 (cit. on p. 1).
- [30] Eric Swanson. *GPU-based Onion Hash generator*. URL: <https://github.com/lachesis/scallion> (cit. on p. 8).
- [31] *The Invisible Internet Project*. URL: <https://geti2p.net> (cit. on p. 2).
- [32] The Tor Project. *Collector – Your friendly data-collecting service in the Tor network*. URL: <https://collector.torproject.org/> (cit. on p. 4).
- [33] The Tor Project. *Tor Metrics*. URL: <https://metrics.torproject.org> (cit. on p. 14).
- [34] Kurt Thomas, Chris Grier, and Vern Paxson. “Adapting Social Spam Infrastructure for Political Censorship”. In: *LEET*. USENIX, 2012. URL: <https://nymity.ch/sybilhunting/pdf/Thomas2012a.pdf> (cit. on p. 1).
- [35] Liang Wang and Jussi Kangasharju. “Real-World Sybil Attacks in BitTorrent Mainline DHT”. In: *Globecom*. IEEE, 2012. URL: <https://nymity.ch/sybilhunting/pdf/Wang2012a.pdf> (cit. on p. 1).
- [36] Philipp Winter. *zoossh – Parsing library for Tor-specific data formats*. URL: <https://gitweb.torproject.org/user/phw/zoossh.git/> (cit. on p. 4).
- [37] Philipp Winter et al. “Spoiled Onions: Exposing Malicious Tor Exit Relays”. In: *PETS*. Springer, 2014. URL: <https://nymity.ch/sybilhunting/pdf/Winter2014a.pdf> (cit. on pp. 1, 3, 4).
- [38] Haifeng Yu, Phillip B. Gibbons Michael Kaminsky, and Feng Xiao. “SybilLimit: A Near-Optimal Social Network Defense against Sybil Attacks”. In: *Security & Privacy*. IEEE, 2008. URL: <https://nymity.ch/sybilhunting/pdf/Yu2008a.pdf> (cit. on p. 2).
- [39] Haifeng Yu et al. “SybilGuard: Defending Against Sybil Attack via Social Networks”. In: *SIGCOMM*. ACM, 2006. URL: <https://nymity.ch/sybilhunting/pdf/Yu2006a.pdf> (cit. on p. 2).

A Exposed malicious exit relays

Table 5 provides an overview of our second dataset, 251 bad exit relays that we discovered between August 2014 and January 2016. We believe that all single relays in the dataset were isolated incidents while sets of relays constituted Sybil groups. Sybil groups marked with the symbols *, †, and ‡ were run by the same attacker, respectively.

Discovery	# of relays	Attack description
Aug 2014	1	The relay injected JavaScript into returned HTML. The script embedded another script from the domain fluxx.crazytall.com—not clearly malicious, but suspicious.
	1	The relay injected JavaScript into returned HTML. The script embedded two other scripts, jquery.js from the official jQuery domain, and clr.js from adobe.flashdst.com. Again, this was not necessarily malicious, but suspicious.
Sep 2014	1	The exit relay routed traffic back into the Tor network, i.e., we observed traffic that was supposed to exit from relay <i>A</i> , but came from relay <i>B</i> . The system presented by Ling et al. behaves the same [23]; the authors proposed to run intrusion detection systems on Tor traffic by setting up an exit relay that runs an NIDS system, and routes the traffic back into the Tor network after having inspected the traffic.
Oct 2014	1	The relay injected JavaScript into returned HTML.
	1	The relay ran the MitM tool sslstrip [25], rewriting HTTPS links to unencrypted HTTP links in returned HTML.
	1	Same as above.
Jan 2015	23*	Blockchain.info’s web server redirects its users from HTTP to HTTPS. These relays tampered with blockchain.info’s redirect and returned unprotected HTTP instead—presumably to sniff login credentials.
	1	The relay used OpenDNS as DNS resolver and had the website category “proxy/anonymizer” blocked, resulting in several inaccessible websites, including torproject.org.
Feb 2015	1	The relay injected a script that attempted to load a resource from the now inaccessible torclick.net. Curiously, torclick.net’s front page said “We place your advertising materials on all websites online. Your ads will be seen only for anonymous network TOR [sic] users. Now it is about 3 million users. The number of users is always growing.”
	17†	Again, these relays tampered with HTTP redirects of Bitcoin websites. Interestingly, the attack became more sophisticated; these relays would begin to target only connections whose HTTP headers resembled Tor Browser.
Mar 2015	18*	Same as above.
	1	The relay injected JavaScript and an iframe into the returned HTML. The injected content was not clearly malicious, but suspicious.
Apr 2015	70‡	These exit relays transparently rewrote onion domains in returned HTML to an impersonation domain. The impersonation domain looked identical to the original, but had different Bitcoin addresses. We believe that this was attempt to trick Tor users into sending Bitcoin transactions to phishing addresses.
Jun 2015	55†	Same as above.
Aug 2015	4†	Same as above.
Sep 2015	1	The relay injected an iframe into returned HTML that would load content that made the user’s browser participate in some kind of mining activity.
Nov 2015	1	The relay ran the MitM tool sslstrip.
	8†	Same as the relays marked with a †.
Dec 2015	1‡	The relay ran the MitM tool sslstrip.
	1‡	Same as above.
Jan 2016	43†	Same as the relays marked with a †.

Table 5: An overview of our second dataset, 251 malicious exit relays that we discovered using exitmap. We believe that Sybil groups marked with an *, †, and ‡ were run by the same adversary.

***k*-fingerprinting: a Robust Scalable Website Fingerprinting Technique**

Jamie Hayes

University College London

j.hayes@cs.ucl.ac.uk

George Danezis

University College London

g.danezis@ucl.ac.uk

Abstract

Website fingerprinting enables an attacker to infer which web page a client is browsing through encrypted or anonymized network connections. We present a new website fingerprinting technique based on random decision forests and evaluate performance over standard web pages as well as Tor hidden services, on a larger scale than previous works. Our technique, *k*-fingerprinting, performs better than current state-of-the-art attacks even against website fingerprinting defenses, and we show that it is possible to launch a website fingerprinting attack in the face of a large amount of noisy data. We can correctly determine which of 30 monitored hidden services a client is visiting with 85% true positive rate (TPR), a false positive rate (FPR) as low as 0.02%, from a world size of 100,000 unmonitored web pages. We further show that error rates vary widely between web resources, and thus some patterns of use will be predictably more vulnerable to attack than others.

1 Introduction

Traditional encryption obscures only the content of communications and does not hide metadata such as the size and direction of traffic over time. Anonymous communication systems obscure both content and metadata, preventing a passive attacker from observing the source or destination of communication.

Anonymous communications tools, such as Tor [11], route traffic through relays to hide its ultimate destination. Tor is designed to be a low-latency system to support interactive activities such as instant messaging and web browsing, and does not significantly alter the shape of network traffic. This allows an attacker to exploit information leaked via the order, timing and volume of resources requested from a website. As a result, many works have shown that website fingerprinting attacks are possible even when a client is browsing with encryption or using an anonymity tool such as Tor [7, 16, 17, 21, 23, 27, 32, 36, 38, 39].

Website fingerprinting is commonly formulated as a classification problem. An attacker wishes to know whether a client browses one of n web pages. The attacker first collects many examples of traffic traces from each of the n web pages by performing web-requests through the protection mechanism under attack; features are extracted and a machine learning algorithm is trained to classify the website using those features. When a client browses a web page, the attacker passively collects the traffic, passes it in to their classifier and checks if the client visited one of the n web pages. In the literature this is referred to as the closed-world setting – a client is restricted to browse a limited number of web pages, monitored by the attacker. However, the closed-world model has been criticized for being unrealistic [17, 29] since a client is unlikely to only browse a limited set of web pages. The open-world setting attempts to model a more realistic setting where the attacker monitors a small number of web pages, but allows a client to additionally browse to a large world size of unmonitored web pages.

Our attack is based on random decision forests [6], an ensemble method using multiple decision trees. We extend the random forest technique to allow us to extract fingerprints to perform identification in an open-world.

The key contributions of this work are as follows:

- A new attack, *k*-fingerprinting, based on extracting a fingerprint for a web page via a novel variant of random forests. We show *k*-fingerprinting is more accurate and faster than other state-of-the-art website fingerprinting attacks [7, 28, 39] even under proposed website fingerprinting defenses.
- An analysis of the features used in this and prior work to determine which yield the most information about an encrypted or anonymized web page. We show that simple features such as counting the number of packets in a sequence leaks more information about the identity of a web page than complex features such as packet ordering or packet inter-arrival time features.
- An open-world setting that is an order of magnitude

larger than the previous open-world website fingerprinting work of 5,000 unmonitored web pages [39]¹, and nearly twice as large in terms of unique numbers websites than [28], reflecting a more realistic website fingerprinting attack over multiple browsing sessions. In total we tested k -fingerprinting on 101,130 unique websites².

- We show that a highly accurate attack can be launched by training a small fraction of the total data, greatly reducing the start-up cost an attacker would need to perform the attack.
- We observe that the error rate is uneven and so it may be advantageous to throw away some training information that could confuse a classifier. An attacker can learn the error rate of their attack from the training set, and use this information to select which web pages they wish to monitor in order to minimize their error rates.
- We confirm that browsing over Tor does not provide any additional protection against fingerprinting attacks over browsing using a standard web browser. Furthermore we show that k -fingerprinting is highly accurate on Tor hidden services, and that they can be distinguished from standard web pages.

2 Related Work

Website Fingerprinting. Website fingerprinting has been studied extensively. Early work by Wagner and Schneier [34], Cheng and Avnur [10] exposed the possibility that encrypted HTTP GET requests may leak information about the URL, conducting preliminary experiments on a small number of websites. They asked clients in a lab setting to browse a website for 5-10 minutes, pausing two seconds between page loading. With caching disabled they were able to correctly identify 88 pages out of 92 using simple packet features. Early website fingerprinting defenses were usually designed to safeguard against highly specific attacks. In 2009, Wright et al. [40] designed ‘traffic morphing’ that allowed a client to shape their traffic to look as if it was generated from a different website. They were able to show that this defense does well at defeating early website fingerprinting attacks that heavily relied on exploiting unique packet length features [21, 32].

¹[17] considers an open world size of $\sim 35K$ but only tried to separate monitored pages from unmonitored pages instead of further classifying the monitored pages to the correct website. The authors assume the adversary monitors four pages: google.com, facebook.com, wikipedia.org and twitter.com. They trained a classifier using 36 traces for each of the Alexa Top 100 web pages, including the web pages of the monitored pages. The four traces for each of the monitored sites plus one trace for each of the unmonitored sites up to $\sim 35K$ are used for testing.

²All code will be made available through code repositories under a liberal open source license and data will be deposited in open data repositories.

In a similar fashion, Tor pads all packets to a fixed-size cells of 512 bytes. Tor also implemented randomized ordering of HTTP pipelines [30] in response to the attack by Panchenko et al. [27] who used packet ordering features to train an SVM classifier. This attack on Tor achieved an accuracy of 55%, compared to a previous attack that did not use such fine grained features achieving 3% accuracy on the same data set using a Naive-Bayes classifier [16]. Other defenses such as the decoy defense [27] loads a camouflage website in parallel to a legitimate website, adding a layer of background noise. They were able to show using this defense attack accuracy of the SVM again dropped down to 3%.

Luo et al. [24] designed the HTTPOS fingerprinting defense at the application layer. HTTPOS acts as a proxy accepting HTTP requests and obfuscating them before allowing them to be sent. It modifies network features on the TCP and HTTP layer such as packet size, packet time and payload size, along with using HTTP pipelining to obfuscate the number of outgoing packets. They showed that HTTPOS was successful in defending against a number of classifiers [5, 9, 21, 32].

More recently Dyer et al. [12] created a defense, BuFLO, that combines many previous countermeasures, such as fixed packet sizes and constant rate traffic. Dyer et al. showed this defense improved upon other defenses at the expense of a high bandwidth overhead. Cai et al. [8] made modifications to the BuFLO defense based on rate adaptation again at the expense of a high bandwidth overhead. Following this Nithyanand et al. [25] proposed Glove, that groups website traffic into clusters that cannot be distinguished from any other website in the set. This provides information theoretic privacy guarantees and reduces the bandwidth overhead by intelligently grouping web traffic in to similar sets.

Cai et al. [7] modified the kernel in Panchenko et al.’s SVM to improve an attack on Tor, and was further improved in an open-world setting by Wang and Goldberg in 2013 [38], achieving a true positive rate (TPR) of over 0.95 and a false positive rate (FPR) of 0.002 when monitoring one web page. Wang et al. [39] conducted attacks on Tor using large open-world sets. Using a k -nearest neighbor classifier they achieved a TPR of 0.85 and FPR of 0.006 when monitoring 100 web pages out of 5100 web pages. More recently Wang and Goldberg [37] suggested a defense using a browser in half-duplex mode – meaning a client cannot send multiple requests to servers in parallel. In addition to this simple modification they add random padding and show they can even foil an attacker with perfect classification accuracy with a comparatively (to other defenses) small bandwidth overhead. Wang and Goldberg [36] then investigated the practical deployment of website fingerprinting attacks on Tor. By maintaining an up-to-date training set and splitting a full

packet sequence in to components comprising of different web page load traces they show that practical website fingerprinting attacks are possible. By considering a time gap of 1.5 seconds between web page loads, their splitting algorithm can successfully parse a single packet sequence in to multiple packet sequences with no loss in website fingerprinting accuracy. Gu et al. [15] studied website fingerprinting in multi-tab browsing setting. Using a Naive Bayes classifier on the 50 top ranked websites in Alexa, they show when tabs are opened with a delay of 2 seconds, they can classify the first tab with 75.9% accuracy, and the background tab with 40.5%. More recently, Kwon et al. [19] showed that website fingerprinting attacks can be applied to Tor hidden services, and due to the long lived structure of hidden services, attacks can be even more accurate than when compared to non-hidden pages. They correctly deanonymize 50 monitored hidden service servers with TPR of 88% and FPR of 7.8% in an open world setting. We further improve on this in our work, resulting in a more accurate attack on the same data set.

In concurrent work, Panchenko et al. [28] have experimented with large datasets. Using a mix of different sources they produced two datasets, one of 34,580 unique websites (118,884 unique web pages) and another of 65,409 unique websites (211,148 unique web pages). Using a variation of a sequence of cumulative summations of packet sizes in a traffic trace they show their attack, CUMUL, was of similar accuracy to k -NN [39] under normal browsing conditions. However, we show that due to their feature set dependency on order and packet counting, their attack suffers substantially under simple website fingerprinting defenses and is outperformed by our technique, k -fingerprinting.

Random Forests. Random forests are a classification technique consisting of an ensemble of decision trees, taking a consensus vote of how to classify a new object. They have been shown to perform well in classification, regression [6, 20] and anomaly detection [22]. Each tree in the forest is trained using labeled objects represented as feature vectors of a fixed size. Training includes some randomness to prevent over-fitting: the training set for each tree is sampled from the available training set with replacement. Due to the bootstrap sampling process there is no need for k -fold cross validation to measure k -fingerprinting performance, it is estimated via the unused training samples on each tree [6]. This is referred to as the *out-of-bag* score.

3 Attack Design

We consider an attacker that can passively collect a client’s encrypted or anonymized web traffic, and aims to infer which web resource is being requested. Dealing with an open-world, makes approaches based purely on

classifying previously seen websites inapplicable. Our technique, k -fingerprinting, aims to define a distance-based classifier. It automatically manages unbalanced sized classes and assigns meaningful distances between packet sequences, where close-by ‘fingerprints’ denote requests likely to be for the same resources.

3.1 k -fingerprints from random forests

In this work we use random forests to extract a fingerprint for each traffic instance³, instead of using directly the classification output of the forest. We define a distance metric between two traces based on the output of the forest: given a feature vector each tree in the forest associates a leaf identifier with it, forming a vector of leaf identifiers for the item, which we refer to as the *fingerprint*.

Once fingerprint vectors are extracted for two traces, we use the Hamming⁴ distance to calculate the distance between these fingerprints⁵. We classify a test instance as the label of the closest k training instances via the Hamming distance of fingerprints – assuming all labels agree. We evaluate the effect of varying k , the number of fingerprints used for comparison, in Sections 7, 8 and 9.

This leafs vector output represents a robust fingerprint: we expect similar traffic sequences are more likely to fall on the same leaves than dissimilar traffic sequences. Since the forest has been trained on a classification task, traces from the same websites are preferentially aggregated in the same leaf nodes, and those from different websites kept apart.

We can vary the number of training instances k a fingerprint should match, to allow an attacker to trade the true positive rate (TPR) for false positive rate (FPR). This is not possible using the direct classification of the random forest. By using a k closest fingerprint technique for classification, the attacker can choose how they wish to decide upon final classification⁶. For the closed-world setting we do not need the additional fingerprint layer for classification, we can simply use the classification output of the random forest since all classes are balanced and our attack does not have to differentiate between False Positives and False Negatives. For the closed-world setting we measure the mean accuracy of the random forest.

³We define a traffic instance as the network traffic generated via a web page load.

⁴We experimented with using the Hamming, Euclidean, Mahalanobis and Manhattan distance functions and found Hamming to provide the best results.

⁵For example, given the Hamming distance function $d : V \times V \rightarrow \mathbb{R}$, where V is the space of leaf symbols, we expect given two packet sequences generated from loading *google.com*, with fingerprints vectors f_1, f_2 and a packet sequence generated from loading *facebook.com* with fingerprint f_3 , that $d(f_1, f_2) < d(f_1, f_3)$ and $d(f_1, f_2) < d(f_2, f_3)$.

⁶We chose to classify a traffic instance as a monitored page if all k fingerprints agree on the label, but an attacker could choose some other metric such as majority label out of the k fingerprints.

3.2 The k -fingerprinting attack

Our k -fingerprinting attack proceeds in two phases: The attacker chooses which web pages they wish to monitor and captures network traffic generated via loading the monitored web pages and a large number of unmonitored web pages. These target traces for monitored websites, along with some traces for unmonitored websites, are used to train a random forest for classification. Given a packet sequence representing each training instance of a monitored web page, it is converted to a fixed length fingerprint as described in Section 3.1 and stored.

The attacker then passively collects instances of web page loads from a client’s browsing session. A fingerprint is extracted from the newly collected packet sequence. The attacker then computes the Hamming distance of this new fingerprint against the corpus of fingerprints collected during training and is classified as a monitored page if and only if all k fingerprints agree on classification, as described in Section 3.1, otherwise it is classified as an unmonitored page.

We define the following performance measures for the attack:

- **True Positive Rate (TPR).** The probability that a monitored page is classified as the correct monitored page.
- **False Positive Rate (FPR).** The probability that an unmonitored page is incorrectly classified as a monitored page.
- **Bayesian Detection Rate (BDR).** The probability that a page corresponds to the correct monitored page given that the classifier recognized it as that monitored page. Assuming a uniform distribution of pages BDR can be found from TPR and FPR using the formula

$$\frac{TPR \cdot Pr(M)}{(TPR \cdot Pr(M) + FPR \cdot Pr(U))}$$

where

$$Pr(M) = \frac{|\text{Monitored}|}{|\text{Total Pages}|}, \quad Pr(U) = 1 - Pr(M).$$

Ultimately BDR indicates the practical feasibility of the attack as it measures the main concern of the attacker, the probability that the classifier made a correct prediction.

4 Data gathering

We collect two data sets: one via Tor⁷ DS_{Tor} , and another via a standard web browser, DS_{Norm} . DS_{Norm} consists of 30 instances from each of 55 monitored web pages, along with 7,000 unmonitored web pages chosen from Alexa’s top 20,000 web sites [1]. We collected DS_{Norm} using a number of Amazon EC2

⁷The most recent version at the time of collection was used, Tor Browser 5.0.6.

instances⁸, Selenium⁹ and the headless browser PhantomJS¹⁰. We used `tcpdump`¹¹ to collect network traces for 20 seconds with 2 seconds between each web page load. Monitored pages were collected in batches of 30 and unmonitored web pages were collected successively. Page loading was performed with no caches and time gaps between multiple loads of the same web page, as recommended by Wang and Goldberg [38]. We chose to monitor web pages from Alexa’s top 100 web sites [1] to provide a comparison with the real world censored web pages used in the Wang et al. [39] data set¹². DS_{Tor} was collected in a similar manner to DS_{Norm} but was collected via the Tor browser. DS_{Tor} consists of two subsets of monitored web pages: (i) 100 instances from each of the 55 top Alexa monitored web pages and (ii) 80 instances from each of 30 popular Tor hidden services. A Tor hidden service is a website that is hosted behind a Tor client’s *Onion Proxy*, which serves as the interface between application and network. Tor hidden services allow both a client accessing the website and the server hosting the website to remain anonymous to one another and any external observers. We chose hidden services to fingerprint based on popularity as listed by the `.onion` search engine Ahmia¹³. The unmonitored set is comprised of the top 100,000 Alexa web pages, excluding the top 55. We chose to fingerprint web pages as listed by Alexa as these constitute the most popular web pages in the world over extended periods of time, and hence provide a more realistic dataset than choosing pages at random and/or using transiently popular website links as included in Panchenko et al.’s recent work [28]. By including website visits to trending topics we argue that this diminishes the ability to properly measure how effective a website fingerprinting attack will perform in general.

For comparison to previous work, we evaluated our attack on one of the previous largest website fingerprinting data sets [39], which collected 90 instances from each of 100 monitored sites, along with 5000 unmonitored web pages. The Wang et al. monitored web pages are various real-world censored websites from UK, Saudi Arabia and China providing a realistic set of web pages an attacker may wish to monitor. The unmonitored web pages are chosen at random from Alexa’s top 10,000 websites – with no intersection between monitored and unmonitored web pages.

⁸<https://aws.amazon.com/ec2/>

⁹<http://www.seleniumhq.org/>

¹⁰<http://phantomjs.org/>

¹¹<http://www.tcpdump.org/>

¹²We used TCP/IP packets for final classification over abstracting to the Tor cell layer [38], preliminary experiments showed no consistent improvements from using one data layer for classification over the other.

¹³<http://www.ahmia.fi/>

This allows us to validate k -fingerprinting on two different data sets while allowing for direct comparison against the state-of-the-art k -Nearest Neighbor attack [39]. We can also infer how well the attack works on censored web pages which may not have small landing pages or be set up for caching like websites in the top Alexa list. Testing k -fingerprinting on both real-world censored websites and top alexa websites indicates how the attack performs across a wide range of websites.

For the sake of comparison, according to a study by research firm Nielsen [3] the number of unique websites visited per month by an average client in 2010 was 89. Another study [17, 26] collected web site statistics from 80 volunteers in a virtual office environment. Traffic was collected from each volunteer for a total of 40 hours. The mean unique number of websites visited per volunteer was 484, this is substantially smaller than the world sizes we consider in our experiments.

5 Feature selection

Our first contribution is a systematic analysis of feature importance. Despite some preliminary work by Panchenko et al. [27], there is a notable absence of feature analysis in the website fingerprinting literature. Instead features are picked based on heuristic arguments. All feature importance experiments were performed with the Wang et al. data set [39] so as to allow direct comparison with their attack results.

We train a random forest classifier in the closed-world setting using a feature vector comprised of features in the literature, and labels corresponding to the monitored sites. We use the gini coefficient as the purity criterion for splitting branches and estimate feature importance using the standard methodology described by Breiman [2, 6, 13]. Each time a decision tree branches on a feature the weighted sum of the gini impurity index for the two descendant nodes is higher than the purity of the parent node. We add up the gini decrease for each individual feature over the entire forest to get a consistent measure of feature importance.

Figure 1 illustrates the effect of using a subset of features for random forest classification. We first train a random forest classifier to establish feature importance; and then train new random forests with only subsets of the most informative features in batches of five. As we increase the number of features we observe a monotonic increase in accuracy; however there are diminishing returns as we can achieve nearly the same accuracy after using the 30 most important features. We chose to use 150 features in all following experiments since the difference in training time when using fewer features was negligible.

Figure 2 identifies the top-20 ranked features and illustrates their variability across 100 repeated experiments.

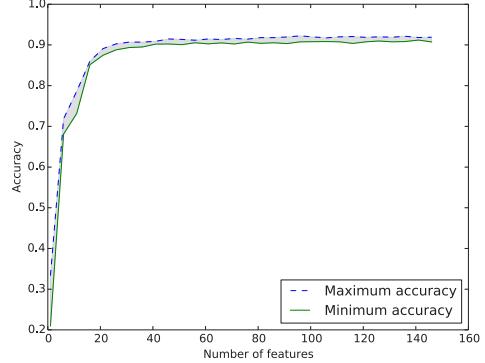


Figure 1: Accuracy of k -fingerprinting in a closed-world setting as the number of features is varied.

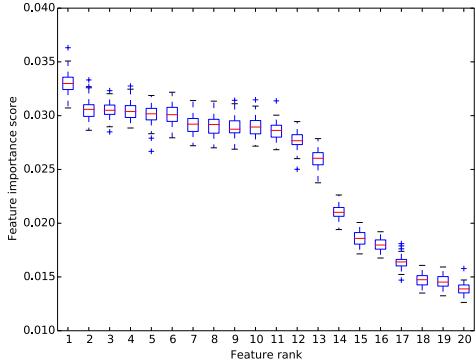
As seen in Figure 1 there is a reduction in gradient when combining the top 15 features compared to using the top 10 features. Figure 2 shows that the top 13 features are comparatively much more important than the rest of the top 20 features, hence there is only a slight increase in accuracy when using the top 15 features compared to using the top 10. After the drop between the rank 13 and rank 14 features, feature importance falls steadily until feature rank 40, after which the reduction in feature importance is less prominent¹⁴. Note that there is some interchangeability in rank between features, we assign ranks based on the average rank of a feature over the 100 experiments.

Feature Importance

From each packet sequence we extract the following features:

- **Number of packets statistics.** The total number of packets, along with the number of incoming and outgoing packets [12, 27, 39]. The number of incoming packets during transmission is the most important feature, and together with the number of outgoing packets during transmission are always two of the five most important features. The total number of packets in transmission has rank 10.
- **Incoming & outgoing packets as fraction of total packets.** The number of incoming and outgoing packets as a fraction of the total number of packets [27]. Always two of the five most important features.
- **Packet ordering statistics.** For each successive incoming and outgoing packet, the total number of packets seen before it in the sequence [7, 27, 39]. The standard deviation of the outgoing packet ordering list has rank 4, the average of the outgoing packet ordering list has rank 7. The standard deviation of the incoming packet ordering list has rank 12 and the average of the

¹⁴The total feature importance table is shown in Appendix A.



Nº	Feature Description
1.	Number of incoming packets.
2.	Number of outgoing packets as a fraction of the total number of packets.
3.	Number of incoming packets as a fraction of the total number of packets.
4.	Standard deviation of the outgoing packet ordering list.
5.	Number of outgoing packets.
6.	Sum of all items in the alternative concentration feature list.
7.	Average of the outgoing packet ordering list.
8.	Sum of incoming, outgoing and total number of packets.
9.	Sum of alternative number packets per second.
10.	Total number of packets.
11-18.	Packet concentration and ordering features list.
19.	The total number of incoming packets stats in first 30 packets.
20.	The total number of outgoing packets stats in first 30 packets.

Figure 2: The 20 most important features.

incoming packet ordering list has rank 13.

- **Concentration of outgoing packets.** The packet sequence split into non-overlapping chunks of 20 packets. Count the number of outgoing packets in each of the chunks. Along with the entire chunk sequence, we extract the standard deviation (rank 16), mean (rank 11), median (rank 64) and max (rank 65) of the sequence of chunks. This provides a snapshot of where outgoing packets are concentrated [39]. The features that make up the concentration list are between the 15th and 30th most important features, but also make up the bulk of the 75 least important features.
- **Concentration of incoming & outgoing packets in first & last 30 packets.** The number of incoming and outgoing packets in the first and last 30 packets [39]. The number of incoming and outgoing packets in the first thirty packets has rank 19 and 20, respectively. The number of incoming and outgoing packets in the last thirty packets has rank 50 and 55, respectively.
- **Number of packets per second.** The number of packets per second, along with the mean (rank 44), standard deviation (rank 38), min (rank 117), max (42), median (rank 50).
- **Alternative concentration features.** This subset of features is based on the concentration of outgoing

packets feature list. The outgoing packets feature list split into 20 evenly sized subsets and sum each subset. This creates a new list of features. Similarly to the concentration feature list, the alternative concentration feature list are regularly in the top 20 most important features and bottom 50 features. Note though concentration features are never seen in the top 15 most important features whereas alternative concentration features are, – at rank 14 and 15, – so information is gained by summing the concentration subsets.

- **Packet inter-arrival time statistics.** For the total, incoming and outgoing packet streams extract the lists of inter-arrival times between packets. For each list extract the max, mean, standard deviation, and third quartile [5]. These features have rank between 40 and 70.

- **Transmission time statistics.** For the total, incoming and outgoing packet sequences we extract the first, second, third quartile and total transmission time [39]. These features have rank between 30 and 50. The total transmission time for incoming and outgoing packet streams are the most important out of this subset of features.

- **Alternative number of packets per second features.** For the number of packets per second feature list we create 20 even sized subsets and sum each subset. The sum of all subsets is the 9th most important feature. The features produced by each subset are in the bottom 50 features - with rank 101 and below. The important features in this subset are the first few features with rank between 66 and 78, that are calculated from the first few seconds of a packet sequence.

We conclude that the total number of incoming packets is the most informative feature. This is expected as different web pages have different resource sizes, that are poorly hidden by encryption or anonymization. The number of incoming and outgoing packets as a fraction of the total number of packets are also informative for the same reason.

The least important features are from the padded concentration of outgoing packets list, since the original concentration of outgoing packets lists were of non-uniform size and so have been padded with zeros to give uniform length. Clearly, if most packet sequences have been padded with the same value this will provide a poor criterion for splitting, hence being a feature of low importance. Packet concentration statistics, while making up the bulk of “useless features” also regularly make up a few of the top 30 most important features, they are the first few items that are unlikely to be zero. In other words, the first few values in the packet concentration list do split the data well.

Packet ordering features have rank 4, 7, 12 and 13, indicating these features are a good criterion for classifi-

cation. Packet ordering features exploit the information leaked via the way in which browsers request resources and the end server orders the resources to be sent. This supports conclusions in [7, 39] about the importance of packet ordering features.

We also found that the number of incoming and outgoing packets in the first thirty packets, with rank 19 and 20, were more important than the number of incoming and outgoing packets in the last thirty packets, with rank 50 and 55. In the alternative number packets per second feature list the earlier features were a better criterion for splitting than the later features in the list. This supports claims by Wang et al. [39] that the beginning of a packet sequence leaks more information than the end of a packet sequence. In contrast to Bissias et al. [5] we found packet inter-arrival time statistics, with rank between 40 and 70, only slightly increase the attack accuracy, despite being a key feature in their work.

6 Attack on hardened defenses

For direct comparison we tested our random forest classifier in a closed-world setting on various defenses against the k -NN attack and the more recent CUMUL [28] attack using the Wang et al. data set [39]. Note that most of these defenses require large bandwidth overheads that may render them unusable for the average client. We test against the following defenses:

- **BuFLO [12].** This defense sends packets at a constant size during fixed time intervals. This potentially extends the length of transmission and requires dummy packets to fill in gaps.
- **Decoy pages [27].** This defense loads a decoy page whenever another page is loaded. This provides background noise that degrades the accuracy of an attack. This is essentially a defense that employs multi-tab browsing.
- **Traffic morphing [40].** Traffic morphing shapes a client’s traffic to look like another set of web pages. A client chooses the source web pages that they would like to defend, as well as a set of target web pages that they would like to make the source processes look like.
- **Tamaraw [35].** Tamaraw operates similarly to BuFLO but fixes packet sizes depending on their direction. Outgoing traffic is fixed at a higher packet interval, this reduces overhead as outgoing traffic is less frequent.
- **Adaptive Padding (AP) [18, 31].** AP protects anonymity by introducing traffic in to statistically unlikely delays between packets in a flow. This limits the amount of extra bandwidth required and does not incur any latency costs. AP uses previously computed histograms of inter-arrival packet times from website loads to determine when a dummy packet should be in-

Table 1: Attack comparison under various website fingerprinting defenses.

Defenses	This work	k -NN [39]	CUMUL [28]	Bandwidth overhead (%)
No defense	0.91 ± 0.01	0.91 ± 0.03	0.91 ± 0.04	0
Morphing [40]	0.90 ± 0.03	0.82 ± 0.06	0.75 ± 0.07	50 ± 10
Decoy pages [27]	0.37 ± 0.01	0.30 ± 0.06	0.21 ± 0.02	130 ± 20
Adaptive Padding [31]	0.30 ± 0.04	0.19 ± 0.03	0.16 ± 0.03	54
BuFLO [12]	0.21 ± 0.02	0.10 ± 0.03	0.08 ± 0.03	190 ± 20
Tamaraw [35]	0.10 ± 0.01	0.09 ± 0.02	0.08 ± 0.03	96 ± 9

jected¹⁵. This is currently the favored option if padding were to be implemented in Tor [4].

Table 1 shows the performance of k -fingerprinting against k -NN and CUMUL under various website fingerprinting defenses in a closed-world setting. Under every defense k -fingerprinting is comparable or achieves better results than the k -NN attack and performs significantly better than CUMUL. Note that k -fingerprinting does equally well when traffic morphing is applied compared to no defense. As Lu et al. [23] note, traffic morphing is only effective when the attacker restricts attention to the same features targeted by the morphing process. Our results confirm that attacks can succeed even when traffic morphing is employed. k -fingerprinting also performs nearly 10% better than k -NN when decoy pages are used, which is in effect a marker for how well the attack performs on multi-tab browsing. Due to the dependency of packet length and sequence length features, CUMUL performs substantially worse than the other two attacks under website fingerprinting defenses. Though CUMUL uses a similar number of features and is of similar computational efficiency to k -fingerprinting, simple defenses remove the feature vector patterns between similar web pages used in CUMUL, rendering the attack ineffectual. More generally, any attack that uses a restricted set of features will suffer greatly from a defense that targets those features. k -fingerprinting performs well under defenses due to its feature set that captures traffic information not used in CUMUL such as packet timings and burst patterns. The k -NN attack performs slightly better than CUMUL but requires an order of magnitude more features than both CUMUL and k -fingerprinting. Our attack is both more efficient and more accurate than CUMUL and k -NN under defenses.

7 k -fingerprinting the Wang et al. data set

We first evaluate k -fingerprinting on the Wang et al. data set [39]. This data set was collected over Tor, and thus implements padding of packets to fixed-size cells (512-bytes) and randomization of request orders [30]. Thus

¹⁵As Juarez et al. [18] note, the distribution of histogram bins is dependent on the individual client bandwidth capacity. Optimizing histograms for a large number of clients is an open problem. Here we implement a naive version of AP with one master histogram for all clients.

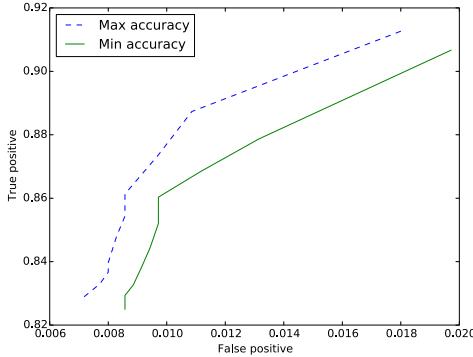


Figure 3: Attack results for 1500 unmonitored training pages while varying the number of fingerprints used for comparison, k , over 10 experiments.

Table 2: k -fingerprinting results for $k=3$ while varying the number of unmonitored training pages.

Training pages	TPR	FPR	BDR
0	0.90 ± 0.02	0.750 ± 0.010	0.419
1500	0.88 ± 0.02	0.013 ± 0.007	0.983
2500	0.88 ± 0.01	0.007 ± 0.001	0.993
3500	0.88 ± 0.01	0.005 ± 0.001	0.997
4500	0.87 ± 0.02	0.009 ± 0.001	0.998

the only available information to k -fingerprinting are timing and volume features. We train on 60 out of the 90 instances for each of the 100 monitored web pages; we vary the number of pages on which we train from the 5000 unmonitored web pages. For the attack evaluation we use fingerprints of length 200 and 150 features. Final classification is as described in Section 3.2, if all k fingerprints agree on classification a test instance is classified as a monitored web page, otherwise it is classified as an unmonitored web page.

The scenario for the attack is as follows: an attacker monitors 100 web pages; they wish to know whether a client is visiting one of those pages, and establish which one. The client can browse to any of these web pages or to 5000 unmonitored web pages, which the attacker classifies in bulk as an unmonitored page.

Using the k -fingerprinting method for classifying a web page we measure a TPR of 0.88 ± 0.01 and a FPR of 0.005 ± 0.001 when training on 3500 unmonitored web pages and k , the number of training instances used for classification, set at $k=3$. k -fingerprinting achieves better accuracy than the state-of-the-art k -NN attack that has a TPR of 0.85 ± 0.04 and a FPR of 0.006 ± 0.004 . Given a monitored web page k -fingerprinting will misclassify this page 12% of the time, while k -NN will misclassify with 15% probability.

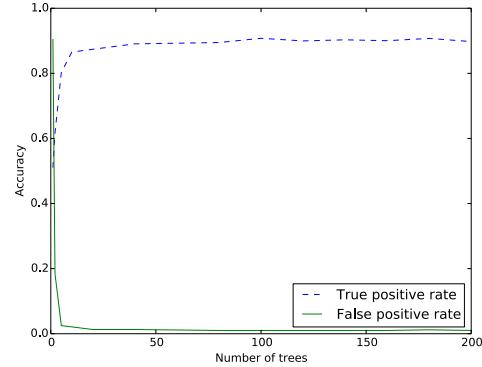


Figure 4: Accuracy of k -fingerprinting as we vary the number of trees in the forest.

Best results are achieved when training on 3500 unmonitored web pages. Table 2 reports TPR and FPR when using different numbers of unmonitored web pages for training with $k=3$. As we train more unmonitored web pages we decrease our FPR with almost no reduction in TPR. After training 3500 unmonitored pages there is no decrease in FPR and so no benefit in training more unmonitored web pages. This is confirmed by the marginal increase in BDR after training on at least some of the unmonitored set. Furthermore without training on any of the unmonitored web pages, despite the high FPR the classifier has more than 40% probability of being correct when classifying a web page as monitored.

Figure 3 illustrates how classification accuracy changes as, k , the number of fingerprints used for classification changes. For a low k the attack achieves a FPR of around 1%, as we increase the value of k we reduce the number of misclassifications since it is less likely that all k fingerprints will belong to the same label, but we also reduce the TPR. Altering the number of fingerprints used for classification allows an attacker to tune the classifier to either a low FPR or high TPR depending on the desired application of the attack.

We find that altering the number of fingerprints used for classification, k , affects the TPR and FPR more than the number of unmonitored training pages. This suggests that while it is advantageous to have a large world size of unmonitored pages, increasing the number of unmonitored training pages does not increase accuracy of the classifier dramatically. This supports Wang et al.’s [39] claims to the same effect. In practice an attacker will need to train on at least some unmonitored pages to increase the BDR, though this does not need to be a substantial amount; training 1500 unmonitored web pages leads to a 98.3% chance the classifier is correct when claiming to have recognized a monitored web page.

Fingerprint length. Changing the length of the fingerprint vector will affect k -fingerprinting accuracy. For a

Table 3: Attack results on top Alexa sites for $k=2$ while varying the number of unmonitored training pages.

Training pages	TPR	FPR	BDR
2000	0.93 ± 0.03	0.032 ± 0.010	0.33
4000	0.93 ± 0.01	0.018 ± 0.007	0.47
8000	0.92 ± 0.01	0.008 ± 0.002	0.67
16000	0.91 ± 0.02	0.003 ± 0.001	0.86

small fingerprint length there may not be enough diversity to provide an accurate measure of distance over all packet sequences. Figure 4 shows the resulting TPR and FPR as we change the length of fingerprints in the Wang et al. [39] data set. We set $k=1$ and train on 4000 unmonitored web pages. Using only a fingerprint of length one results in a TPR of 0.51 and FPR of 0.904. Clearly using a fingerprint of length one results in a high FPR since there is a small universe of leaf symbols from which to create the fingerprint. A fingerprint of length 20 results in a TPR of 0.87 and FPR of 0.013. After this there are diminishing returns for increasing the length of the fingerprint vector.

8 Attack evaluation on DS_{Tor}

We now evaluate k -fingerprinting on DS_{Tor} . First we evaluate the attack given a monitored set of the top 55 Alexa web pages, with 100 instances for each web page. Then we evaluate the attack given a monitored set of 30 Tor hidden services, with 80 instances for each hidden service. The unmonitored set remains the same for both evaluations, the top 100,000 Alexa web pages with one instance for each web page.

8.1 Alexa web pages monitored set

Table 3 shows the accuracy of k -fingerprinting as the number of unmonitored training pages is varied. For the monitored web pages, 70 instances per web page were trained upon and testing was done on the remaining 30 instances of each web page. As expected, the FPR decreases as the number of unmonitored training samples grows. Similar to Section 7 there is only a marginal decrease in TPR while we see a large reduction in the FPR as the number of training samples grows. Meaning an attacker will not have to compromise on TPR to decrease the FPR; when scaling the number of unmonitored training samples from 2% to 16% of the entire set the TPR decreases from 93% to 91% while the FPR decreases from 3.2% to 0.3%. There is a more pronounced shift in BDR with the increase of unmonitored training pages, however an attacker needs to train on less than 10% of the entire dataset to have nearly 70% confidence that classifier was correct when it claims to have detected a monitored page.

Clearly the attack will improve as the number of train-

Table 4: Attack results on Tor hidden services for $k=2$ while varying the number of unmonitored training pages.

Training pages	TPR	FPR	BDR
2000	0.82 ± 0.03	0.0020 ± 0.0015	0.72
4000	0.82 ± 0.04	0.0007 ± 0.0006	0.88
8000	0.82 ± 0.02	0.0002 ± 0.0001	0.96
16000	0.81 ± 0.02	0.0002 ± 0.0002	0.97

ing samples grows, but in reality an attacker may have limited resources and training on a significant fraction of 100,000 web pages may be unfeasible. Figure 5 shows the TPR and FPR of k -fingerprinting as the number of unmonitored web pages used for testing grows while the number of unmonitored web pages used for training is kept at 2000, for different values of k . We may think of this as the evaluation of success of k -fingerprinting as a client browses to more and more web pages over multiple browsing sessions. Clearly for a small k , both TPR and FPR will be comparatively high. Given that, with $k=5$ only 2.5% of unmonitored web pages are falsely identified as monitored web pages, out of 98,000 unmonitored web pages.

8.2 Hidden services monitored set

Table 4 shows the accuracy of k -fingerprinting as the number of unmonitored training pages is varied. For the monitored set, 60 instances per hidden service were trained upon and testing was done on the remaining 20 instances of each hidden service. Again we observe a marginal loss of TPR and a large reduction in FPR as the number of training samples grows. When scaling the number of unmonitored training samples from 2% to 16% of the entire set the TPR decreases from 82% to 81% while the FPR decreases by an order of magnitude from 0.2% to 0.02%. As a result, when training on 16% of the unmonitored set only 16 unmonitored web pages out of 84,000 were misclassified as a Tor hidden service. In comparison to the Alexa web pages monitored set the TPR is around 10% lower, while the FPR is also greatly reduced. This is evidence that Tor hidden services are easy to distinguish from standard web pages loaded over Tor. There is also a higher but more gradual increase in BDR compared to standard web pages. An attacker need only train on as little as 2% of unmonitored pages to have over 70% confidence that classification of a monitored page was correct, with this rising to 97% when training on 16% of the unmonitored dataset.

Similarly to Figure 5, Figure 6 shows the TPR and FPR of k -fingerprinting as the number of unmonitored web pages used for testing grows while the number of unmonitored web pages used for training is kept at 2000, for different values of k . Both the TPR and FPR is lower

than in Figure 5. For example using $k=5$, the FPR is 0.2% which equates to only 196 out of 98,000 unmonitored pages being falsely classified as monitored pages.

From Figure 7 we observe that the BDR of both standard web pages and hidden services monitored sets depends heavily on not only the world size but the number of fingerprints used for classification. With $k=10$, when a web page is classified as a monitored hidden service page, there is over an 80% chance that the classifier was correct, despite the unmonitored world size (98,000) being over 160 times larger than the monitored world size (600). The high BDR regardless of the disparity in world sizes makes it clear that our attack is highly effective under realistic large world size conditions.

It is clear that an attacker need only train on a small fraction of data to launch a powerful fingerprinting attack. It is also clear that Tor hidden services are easily distinguished from standard web pages, rendering them vulnerable to website fingerprinting attacks. We attribute the lower FPR of Tor hidden services when compared to a monitored training set of standard web page traffic to this distinguishability. A standard web page over Tor is more likely to be confused with another standard web page than a Tor hidden service.

Comparison with Kwon et al. [19] hidden services results. For comparison we ran k -fingerprinting on the data set used in the Kwon et al. study on fingerprinting hidden services. This data set simulated a client connecting to a hidden service. The data set consists of 50 instances for each of 50 monitored hidden services and an unmonitored set of 950 hidden services. When training on 100 of the unmonitored pages they report attack accuracy of 0.9 TPR and 0.4 FPR. k -fingerprinting achieved a similar true positive rate but the FPR is reduced to 0.22. This FPR reduction in comparison with Kwon et al. continued regardless of the amount of data used for training.

9 Attack evaluation on DS_{Norm}

Besides testing on DS_{Tor} , Wang et al. [39] data set and the Kwon et al. [19] data set we tested the efficacy of k -fingerprinting on DS_{Norm} . This allows us to establish how accurate k -fingerprinting is over a standard encrypted web browsing session or through a VPN.

9.1 Attack on encrypted browsing sessions

An encrypted browsing session does not pad packets to a fixed size so the attacker may extract the following features in addition to time features:

- **Size transmitted.** For each packet sequence we extract the total size of packets transmitted, in addition, we extract the total size of incoming packets and the total size of outgoing packets.
- **Size transmitted statistics.** For each packet sequence we extract the average, variance, standard deviation

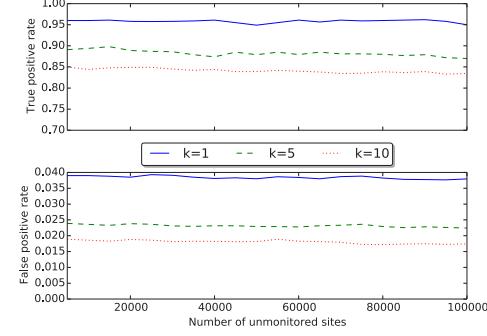


Figure 5: Attack accuracy on DS_{Tor} with Alexa monitored set.

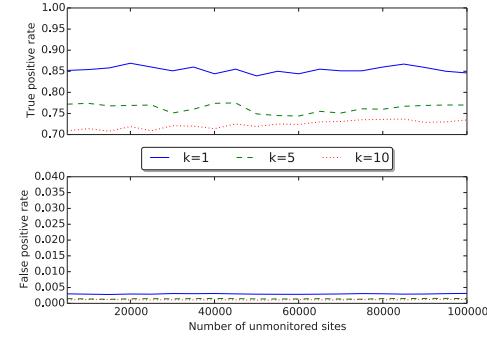


Figure 6: Attack accuracy on DS_{Tor} with Tor hidden services monitored set.

and maximum packet size of the total sequence, the incoming sequence and the outgoing sequence.

Apart from this modification in available features, the attack setting is similar: An attacker monitors a client browsing online and attempts to infer which web pages they are visiting. The only difference is that browsing with the Transport Layer Security (TLS) protocol, or Secure Sockets Layer (SSL) protocol, versions 2.0 and 3.0, exposes the destination IP address and port. The attack is now trying to infer which web page the client is visiting from the known website¹⁶.

The attacker monitors 55 web pages; they wish to know if the client has visited one of these pages. The client can browse to any of these web pages or to 7000 other web pages, which the attacker does not care to classify other than as unmonitored. We train on 20 out of the 30 instances for each monitored page and vary the number of unmonitored pages on which we train.

Despite more packet sequence information to exploit, the larger cardinality of world size gives rise to more

¹⁶Note that the data sets are composed of traffic instances from some websites without SSL and TLS, as well as websites using the protocols. We expect our experiment conditions are much larger than the number possible web pages an attacker may wish to fingerprinting from a standard website.

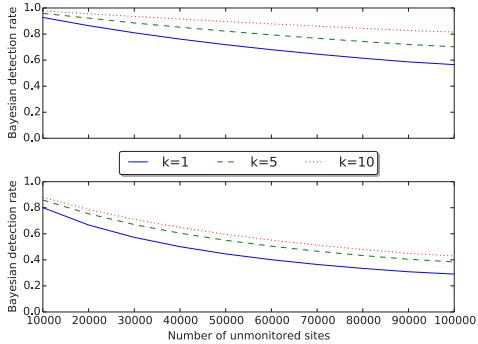


Figure 7: BDR for hidden services monitored set (above) and Alexa monitored set (below).

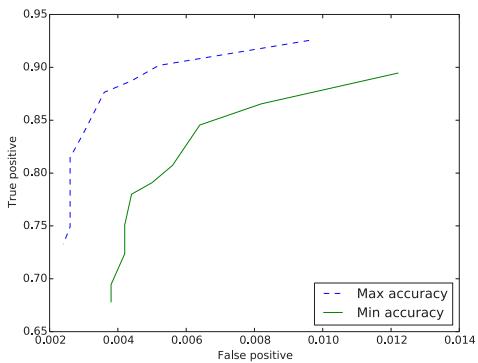


Figure 8: Attack results for 2000 unmonitored training pages while varying the number of fingerprints used for comparison, k , over 10 experiments.

opportunities for incorrect classifications. The attack achieves a TPR of 0.87 and a FPR of 0.004. We achieved best results when training on 4000 unmonitored web pages. Table 5 reports results for training on different numbers of unmonitored web pages, with $k = 2$. Figure 8 shows our results when modifying the number of fingerprints used (k) and training on 2000 unmonitored pages. We find that altering the number of unmonitored training pages decreases the FPR while only slightly decreasing the TPR. This mirrors our experimental findings from DS_{Tor} and the Wang et al. data set.

9.2 Attack without packet size features

DS_{Norm} was not collected via Tor and so also contains packet size information. We remove this to allow for comparison with DS_{Tor} and the Wang et al. data set which was collected over Tor. This also gives us a baseline for how much more powerful k -fingerprinting is when we have additional packet size features available. We achieved a TPR of 0.81 and FPR of 0.005 when training on 5000 unmonitored web pages. Table 6 shows our results at other sizes of training samples, with $k=2$.

Table 5: Attack results with packet size features for a varying number of unmonitored training pages.

Training pages	TPR	FPR	BDR
0	0.95 ± 0.01	0.850 ± 0.010	0.081
2000	0.90 ± 0.01	0.010 ± 0.004	0.908
4000	0.87 ± 0.02	0.004 ± 0.001	0.976
6000	0.86 ± 0.01	0.005 ± 0.002	0.990

Table 6: Attack results without packet size features for a varying number of unmonitored training pages.

Training pages	TPR	FPR	BDR
0	0.90 ± 0.01	0.790 ± 0.020	0.082
2000	0.83 ± 0.01	0.009 ± 0.001	0.910
4000	0.81 ± 0.02	0.006 ± 0.001	0.961
6000	0.80 ± 0.02	0.005 ± 0.001	0.989

Removing packet size features reduces the TPR by over 0.05 and increases the FPR by 0.001. Clearly packet size features improve our classifier in terms of correct identifications but do not decrease the number of unmonitored test instances that were incorrectly classified as a monitored page. Despite the difference in FPR when including packet size information, the BDR is similar, suggesting that BDR is dominated by the amount of information that can be trained upon.

Closed-World. In the closed-world setting in which the client can only browse within the 55 monitored web pages k -fingerprinting is 0.91, compared to 0.96 when packet size features are available. In the closed-world setting attack accuracy improves by 5% when we include packet size features.

10 Fine grained open-world false positives on Alexa monitored set of DS_{Tor}

We observe that the classification error is not uniform across all web pages¹⁷. Some pages are misclassified many times, and confused with many others, while others are never misclassified. An attacker can leverage this information to estimate the misclassification rate of each of the web page classes instead of using the global average misclassification rate. A naive approach to this problem would be to first find which fingerprints contribute to the many misclassifications and remove them. Our analysis shows that the naive approach of removing “bad” fingerprints that contribute to many misclassifications will ultimately lead to a higher misclassification rate. Figure 9 shows the 50 fingerprints that cause the most misclassifications, and also shows for those same fingerprints the number of correct classifications they contribute towards.

¹⁷See additional evidence in Appendix B.

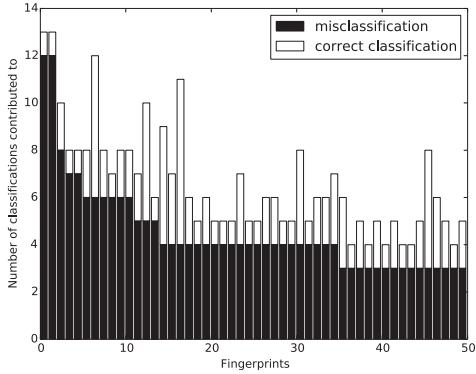


Figure 9: The fingerprints that lead to the most misclassifications and the correct classifications they contribute towards. Training on 2% of unmonitored pages with $k=3$.

As we can see nearly all “bad” fingerprints actually contribute to many correct classifications. One may think it may still be beneficial to remove these fingerprints as the cumulative sum of misclassifications outweigh the number of correct classifications. This removal will then promote fingerprints that are further away in terms of Hamming distance from the fingerprinting that is being tested, which will lead to a greater number of misclassifications.

Instead an attacker can use their training set of web pages to estimate the TPR and FPR of each web page class, by splitting the training set in to a smaller training set and validation set. Since both sets are from the original training set the attacker has access to the true labels. The attacker then computes the TPR and FPR rates of each monitored class, this is used as an estimation for TPR and FPR when training on the entire training set and testing on new traffic instances. More specifically we split, for the monitored training set of 70 instance for each of the Alexa top 55 web pages, into smaller training sets of 40 instances and validation sets of 30 instances. This is used as a misclassification estimator for the full monitored training set against the monitored test set of 30 instances per class. Similarly we split the unmonitored training in half, one set as a smaller training set and the other as a validation set.

Figure 10 shows the TPR and FPR estimation accuracy for 2000 unmonitored training pages. Monitored classes are first ordered from best to worst in terms of their classification accuracy. Even with a small unmonitored training set of 2000 web pages, which is then split in to a training set of 1000 web pages and a validation set of 1000 web pages, an attacker can accurately estimate the FPR of the attack if some of the monitored classes were removed. For example, using only the best 20 monitored classes (in terms of TPR), an attacker would estimate that using those 20 classes as a monitored set, the

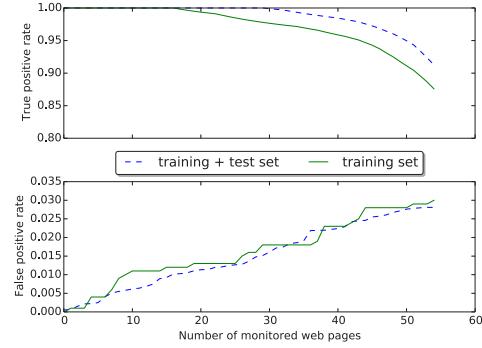


Figure 10: Rates for training on 1000 unmonitored pages, testing on 1000, and comparison when training on the full 2000 unmonitored pages and testing on the remaining 98000 unmonitored pages in DS_{Tor} , $k=3$.

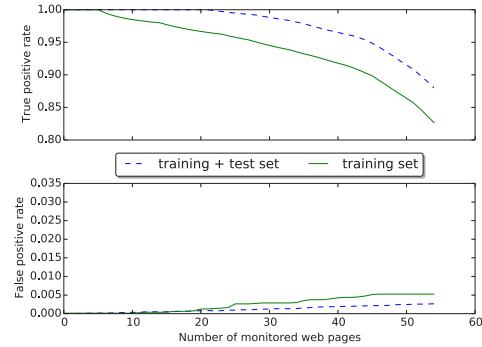


Figure 11: Rates for training on 8000 unmonitored pages, testing on 8000, and comparison when training on the full 16000 unmonitored pages and testing on the remaining 84000 unmonitored pages in DS_{Tor} , $k=3$.

FPR would be 0.012. Using the entire data set we see that the true FPR of these 20 classes is 0.010; the attacker has nearly precisely estimated the utility of removing a large fraction of the original monitored set.

There is a small difference between estimated and the actual FPR in both Figures 10 and 11. There is little benefit in training more unmonitored data if the attacker wants to accurately estimate the FPR; Figure 10 has a similar gap between the estimated FPR and true FPR when compared to Figure 11. It is evident even with a small training set, an attacker can identify web pages that are likely to be misclassified and then accurately calculate the utility of removing these web pages from their monitored set. Due to the overwhelmingly large world size of unmonitored web pages the BDR of Figure 10 does not grow dramatically with the removal of web pages that are likely to be misclassified; using the entire monitored set the BDR is 0.33, removing half of the monitored web pages the BDR is 0.35.

11 Attack Summary & Discussion

Attack Summary. Best attack results on data sets were achieved when training on approximately two thirds of the unmonitored web pages. Despite this, results from DS_{Tor} show that an attacker can achieve a very small false positive rate while only training on 2% of the unmonitored data. Training on 2% of 100,000 unmonitored web pages greatly reduces the attack set up costs while only marginally reducing the accuracy compared to training on more data, providing a realistic setting under which an attack could be launched. Results on all data sets also suggest that altering k , the number of fingerprints used for classification, has a greater influence on accuracy than the number of training samples¹⁸.

k -fingerprinting is robust; our technique achieves the same accuracy regardless of the type of monitored set or the manner in which it was collected (through Tor or standard web browsers). The monitored set in the Wang et al. [39] data set consists of real world censored websites, the Kwon et al. [19] monitored set consist of Tor hidden services and the $DS_{Tor/Norm}$ monitored sets were taken from Tor hidden services and top Alexa websites. We do see a reduction in FPR when the target monitored set are Tor hidden services due to the distinguishability between the hidden services and unmonitored standard web pages.

We also highlight the non-uniformity of classification performance: when a monitored web page is misclassified, it is usually misclassified on multiple tests. We show that an attacker can use their training set to estimate the error rate of k -fingerprinting per web page, and select targets with low misclassification rates.

Computational Efficiency. k -fingerprinting is more accurate and uses fewer features than state-of-the-art attacks. Furthermore k -fingerprinting is faster than current state-of-the-art website fingerprinting attacks. On the Wang et al. data set training time for 6,000 monitored and 2,500 unmonitored training pages is 30.738 CPU seconds on an 1.4 GHz Intel Core i5z. The k -NN attack [39] has training time per round of 0.064 CPU seconds for 2500 unmonitored training pages. For 6,000 rounds training time is 384.0 CPU seconds on an AMD Opteron 2.2 GHz cores. This can be compared to around 500 CPU hours using the attack described by Cai et al. [7]. Testing time per instance for k -fingerprinting is around 0.1 CPU seconds, compared to 0.1 CPU seconds to classify one instance for k -NN and 450 CPU seconds for the attack described by Cai et al. [7].

Discussion. Website fingerprinting research has been criticized for not being applicable to real-world scenarios

[17, 29]. We have shown that a website fingerprinting attack can scale to the number of traffic instance an attacker may sample over long period of time with a high BDR and low FPR. However, we did not consider the cases where background traffic may be present, for example from multitab browsing, or the effect that short-lived websites will have on our attack. Gu et al. [15] show in their work that a simple Naive-Bayes attack achieves highly accurate results even when a client browses in multiple tabs. Wang and Goldberg [36] also show that website fingerprinting is effective in practical scenarios. With no prior attack set-up to tailor to a multi-tab browsing session our attack was able to classify nearly 40% of monitored pages correctly when the decoy defense was employed.

Website content rapidly changes which will negatively affect the accuracy of a website fingerprinting attack [17]. As the content of a website changes so will the generated packet sequences, if an attacker cannot train on this new data then an attack will suffer. However we note that an attack will suffer from the ephemeral nature of websites at different rates depending on the type of website being monitored. For example, an attack monitoring a news or social media site can expect a faster degradation in performance compared to an attack monitoring a landing page of a top 10 Alexa site [1]. Also note Tor does not cache by default, so if in the realistic scenario where an attacker wanted to monitor www.socialmediawebsite.com a client would be forced to navigate to the social media website landing page, which is likely to host content that is long lived and not subject to change. The problem of content change is weakened when fingerprinting Tor hidden services. As show by Kwon et al. [19] hidden pages show minimal changes in comparison to non-hidden pages, resulting in devastatingly accurate attacks on hidden services that can persist.

12 Conclusion

We establish that website fingerprinting attacks are a serious threat to online privacy. Clients of both Tor and standard web browsers are at risk from website fingerprinting attacks regardless of whether they browse to hidden services or standard websites. k -fingerprinting improves on state-of-the-art attacks in terms of both speed and accuracy: current website fingerprinting defenses either do not defend against k -fingerprinting or incur very high bandwidth overheads. Our world size is an order of magnitude larger than previous website fingerprinting studies, and twice as large in terms of unique website than Panchenko et al.’s recent work [28]. We have validated our attack on four separate datasets showing that it is robust and not prone to overfit one dataset, and so is applicable to real world browsing environments at scale. k -fingerprinting is highly accurate even when an attacker

¹⁸Figure 17 illustrates that compared to training on a small number of monitored instances increasing the size of the monitored training set only incrementally increases accuracy.

trains on a small fraction of the total data. Untrustworthy data within that small fraction can then be filtered and removed before the attack is launched to later yield better results, showing that long term website fingerprinting attacks on a targeted client is a realistic threat.

References

- [1] Alexa The Web Information Company, [Accessed August 2015]. URL <http://alexa.com>.
- [2] Leo Breiman. Random Forests, [Accessed July 2015]. URL <https://www.stat.berkeley.edu/~breiman/RandomForests/>.
- [3] The Nielsen Company, [Accessed July 2015]. URL <http://www.nielsen.com/us/en/insights/news/2010/led-by-facebook-twitter-global-time-spent-on-social-media-sites-up-82-year-over-year.html>.
- [4] Tor Proposal 254, [Accessed November 2015]. URL <https://gitweb.torproject.org/torspec.git/tree/proposals/254-padding-negotiation.txt>.
- [5] George Dean Bissias, Marc Liberator, David Jensen, and Brian Neil Levine. "Privacy Vulnerabilities in Encrypted HTTP Streams". In *Proceedings of the 5th International Conference on Privacy Enhancing Technologies*, pages 1–11, 2006.
- [6] Leo Breiman. "Random Forests". *Mach. Learn.*, 45(1):5–32, 2001.
- [7] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. "Touching from a distance: website fingerprinting attacks and defenses". In *ACM Conference on Computer and Communications Security*, pages 605–616, 2012.
- [8] Xiang Cai, Rishab Nithyanand, and Rob Johnson. "CS-BuFLO: A Congestion Sensitive Website Fingerprinting Defense". In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 121–130, 2014.
- [9] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. "Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow". In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 191–206, 2010.
- [10] Heyning Cheng, Heyning Cheng, and Ron Avnur. "Traffic Analysis of SSL Encrypted Web Browsing", 1998.
- [11] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. "Tor: The Second-Generation Onion Router". In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, 2004.
- [12] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. "Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail". In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 332–346, 2012.
- [13] Jerome H. Friedman. "Greedy Function Approximation: A Gradient Boosting Machine". *Annals of Statistics*, 29:1189–1232, 2000.
- [14] Pall Oskar Gislason, Jon Atli Benediktsson, and Johannes R. Sveinsson. "Random Forests for Land Cover Classification". *Pattern Recogn. Lett.*, 27(4):294–300, March 2006.
- [15] Xiaodan Gu, Ming Yang, and Junzhou Luo. "A novel Website Fingerprinting attack against multi-tab browsing behavior". In *19th IEEE International Conference on Computer Supported Cooperative Work in Design, CSCWD*, pages 234–239, 2015.
- [16] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. "Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naive-bayes Classifier". In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, pages 31–42, 2009.
- [17] Marc Juárez, Sadia Afroz, Gunes Acar, Claudia Díaz, and Rachel Greenstadt. "A Critical Evaluation of Website Fingerprinting Attacks". In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 263–274, 2014.
- [18] Marc Juárez, Mohsen Imani, Mike Perry, Claudia Díaz, and Matthew Wright. "WTF-PAD: toward an efficient website fingerprinting defense for tor". *CoRR*, abs/1512.00524, 2015. URL <http://arxiv.org/abs/1512.00524>.
- [19] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. "Circuit Fingerprinting Attacks: Passive Deanonimization of Tor Hidden Services". In *24th USENIX Security Symposium*, pages 287–302, 2015.
- [20] A. Liaw and M. Wiener. "Classification and Regression by randomForest". *R News: The Newsletter of the R Project*, 2(3):18–22, 2002.
- [21] Marc Liberator and Brian Neil Levine. "Inferring the source of encrypted HTTP connections". In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 255–263, 2006.
- [22] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation-Based Anomaly Detection". *ACM Trans. Knowl. Discov. Data*, 6(1):3:1–3:39, March 2012.
- [23] Liming Lu, Ee-Chien Chang, and Mun Choon Chan. "Website Fingerprinting and Identification Using Ordered Feature Sequences". In *Proceedings of the 15th European Conference on Research*

- in Computer Security*, pages 199–214, 2010.
- [24] Xiapu Luo, Peng Zhou, Edmond W. W. Chan, Wenke Lee, Rocky K. C. Chang, and Roberto Perdisci. "HTTPoS: Sealing information leaks with browser-side obfuscation of encrypted flows". In *In Proc. Network and Distributed Systems Symposium (NDSS)*, 2011.
- [25] Rishab Nithyanand, Xiang Cai, and Rob Johnson. "Glove: A Bespoke Website Fingerprinting Defense". In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 131–134, 2014.
- [26] A. Stolerman M. V. Ryan P. Brennan P. Juola, J. I. Noecker Jr and R. Greenstadt. "A Dataset for Active Linguistic Authentication". In *IFIP WG 11.9 International Conference on Digital Forensics*, 2013.
- [27] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. "Website fingerprinting in onion routing based anonymization networks". In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society, WPES*, pages 103–114, 2011.
- [28] Andriy Panchenko, Fabian Lanze, Andreas Zinnen, Martin Henze, Jan Pennekamp, Klaus Wehrle, , and Thomas Engel. "Website Fingerprinting at Internet Scale". In *Network and Distributed System Security Symposium*, 2016.
- [29] Mike Perry. "A Critique of Website Traffic Fingerprinting Attacks". <https://blog.torproject.org/blog/critique-website-traffic-fingerprinting-attacks>, Accessed June 2015.
- [30] Mike Perry. "Experimental defense website traffic fingerprinting". <https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting>, Accessed June 2015.
- [31] Vitaly Shmatikov and Ming-Hsiu Wang. "Timing Analysis in Low-Latency Mix Networks: Attacks and Defenses". In *ESORICS*, 2006.
- [32] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. "statistical identification of encrypted web browsing traffic". In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 19–, 2002.
- [33] Vladimir Svetnik, Andy Liaw, Christopher Tong, J. Christopher Culberson, Robert P. Sheridan, and Bradley P. Feuston. "Random Forest: A Classification and Regression Tool for Compound Classification and QSAR Modeling". *Journal of Chemical Information and Computer Sciences*, 43(6):1947–1958, 2003.
- [34] David Wagner and Bruce Schneier. "Analysis of the SSL 3.0 Protocol". In *Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, pages 4–4, 1996.
- [35] T. Wang and I. Goldberg. "Comparing website fingerprinting attacks and defenses". Technical Report, 2013.
- [36] T. Wang and I. Goldberg. "On Realistically Attacking Tor with Website Fingerprinting". Technical Report, 2015.
- [37] T. Wang and I. Goldberg. "Walkie-Talkie: An Effective and Efficient Defense against Website Fingerprinting". Technical Report, 2015.
- [38] Tao Wang and Ian Goldberg. "Improved Website Fingerprinting on Tor". In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, pages 201–212, 2013.
- [39] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. "Effective Attacks and Provable Defenses for Website Fingerprinting". In *Proceedings of the 23rd USENIX Security Symposium*, pages 143–157, 2014.
- [40] Charles V. Wright, Scott E. Coull, and Fabian Monroe. "Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis". In *In Proceedings of the 16th Network and Distributed Security Symposium*, pages 237–250, 2009.

A Total feature importance.

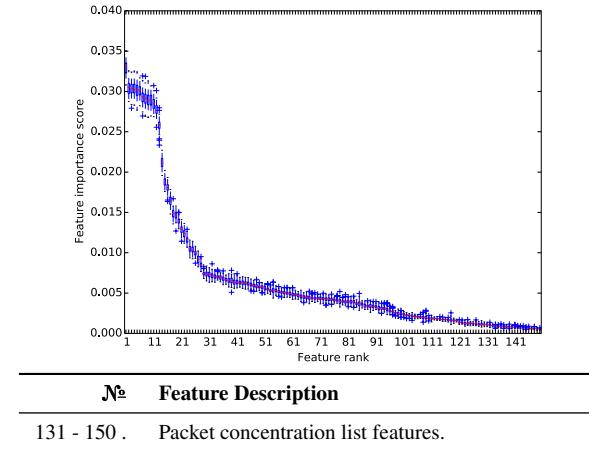


Figure 12: Feature importance score for all 150 features in order. The table gives the description for the 20 least important features.

B Closed World Error Rates

Figure 13 shows the confusion matrix in our closed-world setting, that is, it shows the 49 misclassifications (out of 550). We see that some persistent misclassification patterns of web pages appear, for example web page 54 is classified correctly four times but is misclassified as web page 0 six times. The misclassification rate in Figure 13 is 0.09 but this is the average error rate across all web pages.

Figure 13 shows that the classification error is not uniform across all web pages. Some pages are misclassified many times, and confused with many others, while others are never misclassified. An attacker can leverage this information to estimate the misclassification rate of each web page instead of using the global average misclassification rate.

As in Section 10, an attacker can use their training set of web pages to estimate the misclassification rate of each web page, by splitting the training set in to a smaller training set and validation set. Since both sets are from the original training set the attacker has access to the true labels. The attacker then computes the misclassification rate of each web page, which they can use as an estimation for the misclassification rate when training on the entire training set and testing on new traffic instances.

Figures 14 and 15 show the global misclassification rate for a varying number of monitored pages. Monitored pages are first ordered in terms of the misclassification rate they have, ordered from smallest to largest. From Figure 14, using the Wang et al. data set, we see that if the attacker considers only the top 50% on web pages in terms of per page misclassification rate, the true

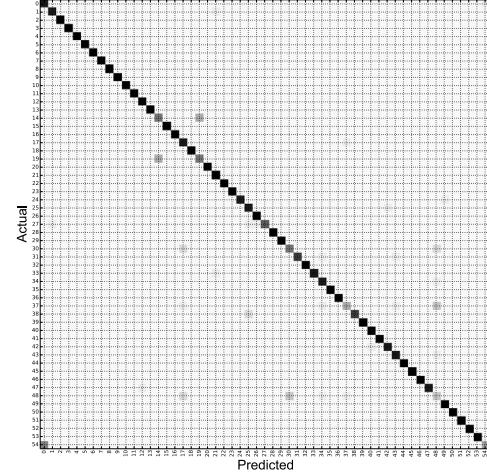


Figure 13: Confusion matrix for closed-world attack on Tor using DS_{Norm} . F1 score = 0.913, Accuracy: 0.915, 550 items.

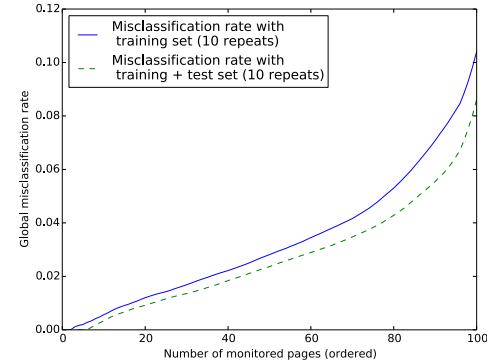


Figure 14: The global misclassification rate when considering different numbers of monitored pages from the Wang et al. data set. The monitored pages are ordered in terms of smallest misclassification rate to largest.

global misclassification rate and global misclassification rate estimated by the attacker drop by over 70%. Similarly from Figure 15, using DS_{Norm} , if the attacker considers only the top 50% on web pages in terms of per page misclassification rate, the true global misclassification rate and global misclassification rate estimated by the attacker drop by over 80%. This allows an attacker to train on monitored pages and then cull the pages that have too high an error rate, allowing for more confidence in the classification of the rest of the monitored pages.

The gap between the attacker's estimate and the misclassification rate of the test set is largely due to the size of the data set. Figure 14 has a smaller error of estimate than Figure 15 because the Wang et al. data set has 60 instances per monitored page, in comparison DS_{Norm} has 20 instances per monitored page. In practice, an attacker

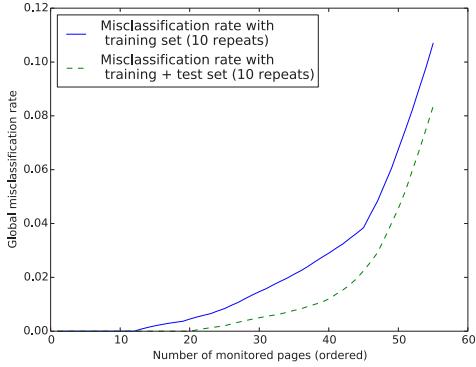


Figure 15: The global misclassification rate when considering different numbers of monitored pages from DS_{Norm} . The monitored pages are ordered in terms of smallest misclassification rate to largest.

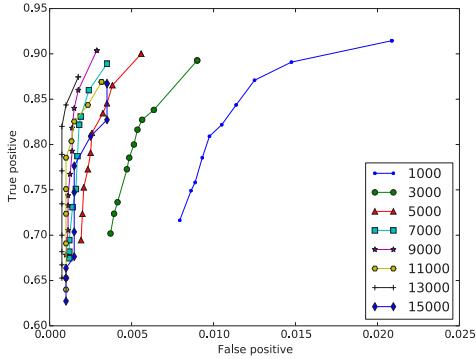


Figure 16: Attack accuracy for 17,000 unmonitored web pages. Each line represents a different number of unmonitored web pages that were trained, while varying k , the number of fingerprints used for classification.

cannot expect perfect alignment; they are generated from two different sets of data, the training and training + test set. Nevertheless the attacker can expect this difference to decrease with the collection of more training instances.

C Attack on larger world size of DS_{Norm}

We run k -fingerprinting on DS_{Norm} with the same number of monitored sites but increase the number of unmonitored sites to 17,000. We evaluate when we have both time and size features available.

Figure 16 shows the results of k -fingerprinting while varying the number of fingerprints (k) used for classification, from between 1 and 10, for various experiments trained with different numbers of unmonitored pages. We see that the attack results are comparable to the attack on 7000 unmonitored pages, meaning there is no degradation in attack accuracy when we increase the world

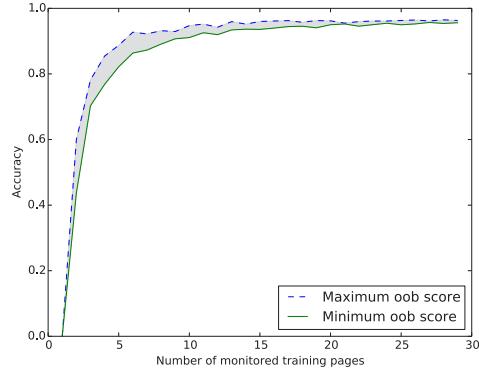


Figure 17: Attack out-of-bag score while varying the number of monitored training pages.

size by 10,000 web pages. Training on approximately 30% of the 7000 unmonitored web pages k -fingerprinting gives a TPR of over 0.90 and a FPR of 0.01 for $k=1$. Training on approximately 30% of the 17,000 unmonitored web pages k -fingerprinting gives a TPR of 0.90 and a FPR of 0.006 for $k=1$.

The fraction of unmonitored pages that were incorrectly classified as a monitored page decreased as we increased our world size. In other words, out of 12,000 unmonitored pages only 72 were classified as a monitored page, with this Figure dropping to 24 if we use $k=10$ for classification. This provides a strong indication that k -fingerprinting can scale to a real-world attack in which a client is free to browse the entire internet, with no decrease in attack accuracy.

Number of monitored training pages in closed-world. Figure 17 shows the *out-of-bag* score as we change the number of *monitored* pages we train. We found that training on any more than a third of the data gives roughly the same accuracy.

Protecting Privacy of BLE Device Users

Kassem Fawaz* Kyu-Han Kim† Kang G. Shin*

*The University of Michigan †Hewlett Packard Labs

Abstract

Bluetooth Low Energy (BLE) has emerged as an attractive technology to enable Internet of Things (IoTs) to interact with others in their vicinity. Our study of the behavior of more than 200 types of BLE-equipped devices has led to a surprising discovery: the BLE protocol, despite its privacy provisions, fails to address the most basic threat of all—hiding the device’s presence from curious adversaries. Revealing the device’s existence is the stepping stone toward more serious threats that include user profiling/fingerprinting, behavior tracking, inference of sensitive information, and exploitation of known vulnerabilities on the device. With thousands of manufacturers and developers around the world, it is very challenging, if not impossible, to envision the viability of any privacy or security solution that requires changes to the devices or the BLE protocol.

In this paper, we propose a new device-agnostic system, called BLE-Guardian, that protects the privacy of the users/environments equipped with BLE devices/IoTs. It enables the users and administrators to control those who discover, scan and connect to their devices. We have implemented BLE-Guardian using Ubertooth One, an off-the-shelf open Bluetooth development platform, facilitating its broad deployment. Our evaluation with real devices shows that BLE-Guardian effectively protects the users’ privacy while incurring little overhead on the communicating BLE-devices.

1 Introduction

Bluetooth Low Energy (BLE) [4] has emerged as the *de facto* communication protocol in the new computing paradigm of the Internet of Things (IoTs) [8, 9, 15, 23, 24, 39]. In 2013, over 1.2 billion BLE products were shipped [9], with this number expected to hit 2.7 billion in 2020 [3]. BLE-equipped products are embedded and used in every aspect of our lives; they sense nearby

objects, track our fitness, control smart appliances and toys provide physical security, etc. The BLE protocol owes this proliferation to its low energy and small processing footprint as well as its support by most end-user devices [20], such as PCs, gateways, smartphones, and tablets.

A BLE-equipped device advertises its presence to let interested nearby devices initiate connections and glean relevant information. These advertisements, however, are a double-edged sword. An unauthorized, potentially malicious, party can use these advertisements to learn more about the BLE-equipped devices of a certain user or in a specific environment [22], generally referred to in literature as the *inventory attack* [42]. Revealing the device’s presence is the stepping stone toward more serious privacy and security attacks with grave consequences in the case of medical devices for example, especially for high-value targets [31].

The BLE specification contains some privacy provisions to minimize the effects of inventory attacks and ensuing threats, namely *address randomization* and *whitelisting*. A BLE device is supposed to randomize its address to prevent others from tracking it over time. Moreover, only devices with a pre-existing trust relationship (whitelisted devices) are supposed to access the BLE-equipped device.

In this paper, we first analyze how existing BLE’s privacy measures fare in the real-world deployments through our own data-collection campaign. To the best of our knowledge, this is the first study that systematically analyzes threats to the BLE-equipped devices in the wild. We recruited participants from our institution and the PhoneLab testbed [27] to collect the BLE advertisements in their vicinity. We have collected and analyzed the advertisements from 214 different types of BLE-equipped devices. Analyzing our dataset has led to a surprising discovery: BLE advertisements, due to poor design and/or implementation, leak an alarming amount of information that allows the tracking, profiling, and

fingerprinting of the users. Furthermore, some devices allow external connections without an existing trust relationship. Unauthorized entities can access unsecured data on the BLE-equipped devices that might leak sensitive information and potentially inflict physical harm to the bearer.

Almost all of the existing approaches addressing some of the above threats rely on mechanisms that necessarily include changes to the protocol itself or to the way the BLE-equipped devices function [21, 40]. Changing the operation of such devices, post-production, requires their patching by securely pushing a firmware update. With thousands of manufacturers and developers around the world, it is very challenging, sometimes impossible, to guarantee firmware patches to the millions of already deployed devices [11]. Even a security-aware user might lack the ability to update the firmware of a BLE-equipped device. Patch management is, therefore, the leading security challenge in the emerging IoTs [10, 19] (including BLE-equipped devices) for many reasons:

- Manufacturers might lack the ability to apply OTA updates [1] for some deployed BLE-equipped devices because they (such as a BLE-equipped pregnancy test) are neither programmable nor equipped with an Internet connection.
- Customers might neither receive news about the update nor be able to apply an update even if available. For example, a month after the 2013 “Foscam” webcams hacking incident, 40,000 of 46,000 vulnerable cameras were not updated although a firmware update was available [17].
- Companies do not have enough financial incentives or resources to maintain the devices post deployment [34]. For example, Samsung discontinued two lines of smart refrigerators after 2012 so that customers can’t receive updates for their purchased refrigerators [6].

There is, therefore, a need for a new class of practical approaches to mitigate the privacy threats to BLE-equipped devices. In this paper, we seek to answer the following related question: *can we effectively fend off the threats to BLE-equipped devices: (1) in a device-agnostic manner, (2) using COTS (Commercial-Off-The-Shelf) hardware only, and (3) with as little user intervention as possible?*

We present BLE-Guardian as an answer to the above question. It is a practical system that protects the user’s BLE-equipped devices so that *only* user-authorized entities can discover, scan, or connect to them. BLE-Guardian relies on an external and off-the-shelf Bluetooth radio as well as an accompanying application. Therefore, a user can easily install (and control) BLE-Guardian to any BLE gateway, be it a smartphone,

tablet, PC, Raspberry PI, Artik-10, etc. The external radio achieves the physical protection, while the application, running on the gateway, enables the user to interact with BLE-Guardian.

BLE-Guardian provides privacy and security protection by targeting the root of the threats, namely the advertisements. In particular, BLE-Guardian opportunistically invokes reactive jamming to determine the entities that can observe the device existence through the advertisements (*device hiding module*), and those that can issue connection requests in response to advertisements (*access control module*). In a typical BLE environment, however, achieving BLE-Guardian’s objective is rather challenging. Many BLE-equipped devices, including the ones to be protected, advertise on the same channel; while at the same time other devices, in response to advertisements, issue scan and connection requests. The timing is of an essence for BLE-Guardian; it invokes jamming at the right time for the right duration. Therefore, BLE-Guardian does not inadvertently harm other devices, preserves the ability of authorized entities to connect the BLE-equipped device, and always hides the BLE-equipped device when needed.

More than one device might be authorized to connect to the BLE-equipped device. BLE-Guardian differentiates the scan and connection requests originating from authorized devices versus those that are fraudulent. This is particularly challenging as the BLE advertisement channel lacks any authentication mechanism for the advertisements and connections. BLE-Guardian utilizes Bluetooth classic as an out-of-band (OOB) channel to authorize a device after obtaining the user’s permission. It uses the OOB channel to instruct the connecting device to issue ordinary connection requests with (varying) special parameters that other unauthorized devices can’t predict. It also alerts the user when unauthorized parties attempt connection to the user’s BLE devices.

BLE-Guardian achieves its objectives with minimum requirements from the external radio. Effectively, BLE-Guardian operates with a radio that offers only the basic capabilities of reception and transmission on the BLE channels. As a result, BLE-Guardian avoids employing sophisticated and customized (thus impractical) radios and signal processing approaches.

We implement BLE-Guardian using the commercially available Ubertooth One¹ USB dongle so that BLE-Guardian can be easily installed on any BLE gateway. We also implement accompanying apps for different BLE gateways, such as Android and Raspberry PI. We evaluate BLE-Guardian using several BLE devices for different real-world scenarios, where we assess its effectiveness in combating privacy threats, its low over-

¹<https://greatscottgadgets.com/ubertoothone/>

head on the channel and devices, and little disruption to the operation of legitimate BLE devices. In particular, BLE-Guardian is able to protect up to 10 class-2 target BLE-equipped devices within a 5m range with less than 16% energy overhead on the gateway.

The rest paper is organized as follows. Section 2 discusses the related work. Section 3 provides the necessary BLE background. Section 4 states the privacy threats arising from BLE advertisements through our data-collection campaign. Section 5 details the design of BLE-Guardian. Section 6 presents the implementation of BLE-Guardian and evaluates its effectiveness. Finally, the paper concludes with Section 7.

2 Related Work

There have been limited efforts related to BLE devices that target the security and privacy threats resulting from the devices revealing their presence. The only exception is the work by Wang [40], where a privacy enhancement is proposed for BLE advertisements to ensure confidentiality and prevent replay attacks as well as tracking. This enhancement is based on providing an additional 3-way handshake between the peripheral and the gateway. Unarguably, this enhancement changes both the protocol and the peripheral which is highly impractical as we argued before.

Another related field of research includes wearable and body-area networks. The work by Leonard [21] uses a honeypot to lure in adversaries that attempt to attack the user’s wearable devices. The honeypot uses a set of helper nodes to expose fake services with known weaknesses so that the attacker connects to them. This work, however, doesn’t handle the privacy threat arising from BLE advertisements. A determined attacker will be able to distinguish fake traffic from legitimate one based on RF signatures from the devices and issue connections to the user’s real devices.

Other relevant work includes approaches to protecting medical devices. Mare *et al.* [25] propose a mechanism that protects health sensors when communicating with a gateway. The proposed system, albeit relevant, doesn’t apply for the BLE ecosystem. It also mandates changing the medical devices. Gollakota *et al.* [12] propose an external device, called *Shield*, that the user wears to control access to his/her embedded medical device. *Shield* implements friendly jamming so that only an authorized programmer can communicate with the medical device.

BLE-Guardian takes an entirely different approach by targeting the control plane of the BLE protocol instead of the data plane. BLE-Guardian does not need to continually protect an ongoing authorized connection and more importantly need not invoke jamming signal cancellation that requires accurate estimation of chan-

nel condition in a dynamic mobile indoor environment as well as a full duplex radio. BLE-Guardian constitutes a reference design that can function with any radio that has reception and transmission capabilities on the 2.4 GHz band. BLE-Guardian, also, considers far less restrictive scenarios than *Shield*. It does not have to be within centimeters of the device-to-be-protected as the case with *Shield*. Moreover, BLE-Guardian’s practical design allows scaling up protection for multiple devices (multiple connectors and protected devices) simultaneously, which is not the case for *Shield* that considers a two-device scenario only [36].

Finally, researchers have explored ways to reduce information leaks from sensors in a smart home environment [30, 37]. Srinivasan *et al.* [37], Park *et al.* [30], and Schurgot *et al.* [35] propose a set of privacy enhancements that include perturbing the timing of broadcasted sensory data along with padding the real sensory data with fake data to confuse the adversary. These protocols fail to address the threats resulting from BLE advertisements and have the shortcoming of requiring changes to the sensors as well.

3 BLE Primer

The BLE (Bluetooth 4.0 and newer) protocol has been developed by the Bluetooth SIG to support low power devices such as sensors, fitness trackers, health monitors, etc. Currently, more than 75,000 devices in the market support this protocol along with most of more capable devices such as smartphones, tablet, PCs, and recently access points [2].

3.1 BLE States

A BLE device assumes either a central or peripheral role. A peripheral device is typically the one with lower capabilities and with the information to advertise. The central device, typically an AP, PC, or smartphone, scans for advertisements and initiates connections.

The BLE specification places a higher burden on the central device. It is responsible for initiating the connection and thus has to keep scanning until it receives an advertisement. Conversely, the peripheral (prior to its connection) sleeps for most of the time and only wakes up to advertise, which helps save its limited energy.

3.2 Advertisements

BLE advertisements are instrumental to the operation of the protocol, and constitute the only means by which a device can be discovered. The specification defines 4 advertisement message types as shown in Table 1, and 3

Table 1: The four types of BLE advertisements.

Type	Advertising Interval
ADV_IND	20ms – 10.24s
ADV_DIRECT_IND	3.75ms
ADV_NONCONN_IND	100ms – 10.24s
ADV_SCAN_IND	100ms – 10.24s

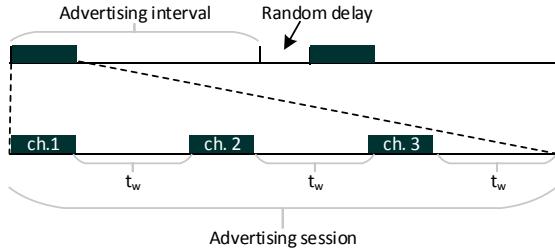


Figure 1: The advertisement pattern in BLE.

advertisement channels: 37 (2402MHz), 38 (2426MHz), and 39 (2480MHz).

ADV_DIRECT_IND (introduced in Bluetooth 4.2) is a special advertisement; it enables fast reconnection between the central and the peripheral devices. The peripheral, when turned on, will broadcast advertisements at a fast rate (once every 3.75ms, for 1.28 seconds) that are directed to the client (with a pre-existing trust relationship) before assuming the central role. The advertisement message only contains the message type, source, and destination addresses.

The other three advertisements are similar to each other in that they are periodic. The advertisement interval determines the frequency with which each device advertises. This interval has to be chosen, at configuration time, between 20ms and 10.24 seconds (at increments of 0.625ms) for the ADV_IND advertisement and between 300ms and 10.24 seconds for the other two advertisements. To prevent advertisements of different devices from colliding with each other, each device waits for a random amount of time between 0 and 10ms (in addition to the advertisement interval) before it advertises (Fig. 1).

The advertisement session constitutes the period when the device is actually advertising. During each advertisement session, the device advertises on the three advertisement channels given a pre-configured channel sequence. Before switching to the next channel, the device has to wait for a preset period of time (less than 10ms) for scan and connection requests (t_w in Fig. 1). We will henceforth refer to the advertisement interval, the channel sequence, and the waiting time as the *advertisement pattern*.

Each advertisement message contains the message

type, source address, along with some of the services offered by the device and their respective values. The specification defines a set of services that have unique UUIDs, such as device name. The message is limited in length, and hence, to get more information about the device, an interested device can either issue a scan request to which the advertising device responds with a scan response or connect to the advertising device.

3.3 Connections

Not all BLE devices accept connections; devices that use ADV_NONCONN_IND advertisement messages run in transmit mode only and they don't accept any scan or connection requests such as iBeacons.

Also, devices advertising with ADV_SCAN_IND messages don't accept connections but accept scan requests. Particularly, when the device broadcasts an advertisement message on some channel, it listens on the same channel for some time (less than 10ms) before switching to the next channel. It waits for scan requests from clients wanting to learn more information to which it responds with a scan response.

Devices that advertise using ADV_IND messages are scannable and connectable; they respond to scan messages and connection requests. After sending an advertisement message, the device listens for connection requests. The connection request contains the source and destination addresses along with other connection parameters. These parameters contain the connection interval, the timeout, and the slave interval. When connected, the device starts frequency hopping according to a schedule negotiated with the central. If the device (now peripheral) doesn't receive any communication from the central over the period defined by the "timeout interval", it drops the connection.

While connected, the device must not broadcast connectable advertisement messages (the first two types of Table 1). It can, however, still broadcast non-connectable advertising messages to share information (the last two types of Table 1) with other clients in its transmission range which still leaks information about the device's name, type, and address.

3.4 Privacy and Security Provisions

The BLE specification borrows some security provisions from classical Bluetooth to establish trust relationships between devices, a process known as *pairing*. When the device boots for the first time, it will advertise using ADV_IND with its public Bluetooth address. The user can then pair a smartphone (or other BLE-equipped device) so that the two devices exchange a secret key that will enable future secure communication.

Once a BLE-equipped device is paired with another device, it can invoke more privacy and security provisions. The first provision is whitelisting, and the device will only accept connections from devices it has been paired with before, i.e., those that are whitelisted. Also, the device might accept connections from any client but might require higher security levels for some of the services it exposes so that only authorized users access sensitive content.

Finally, the BLE specification defines a privacy provision based on address randomization to prevent device tracking. When two devices are paired, they exchange an additional key called the *Identity Resolution Key* (IRK). The device uses this key to generate a random address according to a timer value set by the manufacturer, which it will use to advertise. This random address will be resolved by the paired device using the same key. As this random address is supposed to change regularly, curious parties shouldn't be able to track a BLE-equipped device. Devices that don't utilize address randomization can resort to direct advertising (ADV_DIRECT_IND) to enable fast and private reconnections.

These privacy provisions are akin to those proposed earlier in the context of WiFi networks. Researchers have long identified privacy leaks from the consistent identifiers broadcasted by wireless devices. They proposed privacy enhancements that include randomizing or frequent disposing of MAC addresses [14, 18] and hiding them through encrypting the entire WiFi packets [13]. These enhancements require introducing changes for the client devices.

4 Threats from BLE Devices

While, in theory, BLE's privacy provisions might be effective to thwart threats to the user's privacy, whether or not various manufacturers and developers properly implement them is an entirely different story. In what follows, we investigate how the BLE privacy provisions fare in the wild using a dataset collected in our institution and using the PhoneLab testbed [27] of SUNY Buffalo.

We developed an Android app that collects the content of the advertisement messages. We recruited users from our institution and from the PhoneLab testbed. We didn't collect any personal information about the users and thus obtained non-regulated status from the IRB of our institution. One could view this study as crowdsourcing the analysis of BLE devices; instead of purchasing a limited set of devices and analyzing them, we monitored the behavior of a broad range of devices in the wild. Analyzing the advertisements we collected from 214 different types of devices (sample of these devices shown in Tables 2 and 3), we observed:

Table 2: A sample of devices with revealing names.

Name	Type
LG LAS751M(27:5D)	music streaming
JS00002074	digital pen
ihere	key finder
spacestation	battery/storage extension
Jabra PULSE Smart	smartbulb
DEXCOMRX	Glucose monitor
Clover Printer 0467	printer
Frances's Band ea:9d LE	smartband
Gear Fit (60ED)	activity tracker
Lyve Home-00228	photo storage
Matthias-FUSE	headset
Richelle's Band b2:6a LE	smartband
vivosmart #3891203273	activity tracker
KFDNX	key fob
OTbeat	heart rate monitor
Thermos-4653	Smart Thermos
POWERDRIVER-L10C3	smart power inverter

Table 3: A sample of devices with consistent addresses for more than a day.

Name	Type	Days observed
One	activity tracker	37
Flex	activity tracker	37
Zip	activity tracker	37
Surge	activity tracker	36
Charge	activity tracker	36
Forerunner 920	smartwatch	36
Basis Peak	sleep tracker	25
MB Chronowing	smartwatch	15
dotti	pixel light	7
UP MOVE	fitness tracker	2
GKChain	laptop security	2
Gear S2 (0412)	smartwatch	2
Crazyflie	quadropter	1
Dropcam	camera	1

1. Two advertisement types (ADV_NONCONN_IND and ADV_SCAN_IND) require a fixed address which would enable tracking of a mobile target.
2. Devices that are bonded to the users advertise using ADV_IND messages instead of the more private ADV_DIRECT_IND.
3. Some devices, albeit not expected to do so, use their public Bluetooth addresses in advertisements. This also enables tracking as well as identifying of the device manufacturer.
4. Other devices implement poor address randomization by flipping some bits of the address rendering them ineffective against tracking. This has also been identified in other studies of BLE devices [22].
5. A large number of devices advertise their names (Table 2), revealing sensitive information about them, the user, and the environment. Also, some of the device names contain personal information

or unique identifiers that may enable user tracking.

6. Some devices use a consistent Bluetooth address for long periods of time which renders address randomization ineffective (Table 3). Examples include various types of wristbands (Fitbit Flex, Forerunner 920, etc.), headsets, smartwatches (Apple Watch or Samsung Gear), etc. This has also been identified by Das *et al.* [7], where they analyzed the advertisements of BLE-equipped fitness trackers. Das *et al.* found the fitness trackers constantly advertising with consistent (non-private) BLE addresses. In our experiments, we observed that Flex and One kept the same address for 37 days, so did One and Charge for 30 days.
7. Some devices accept connections from non-bonded devices. This allows access to the services on the device including the unique manufacturer ID, for instance, which allows for user tracking regardless of the device’s address. For example, we were able to connect to various devices and access data from them without any existing trust relationship, such as various Fitbit devices (One, Zip Flex, Charge), Garmin Vivosmart, digital pens, etc. It is worth noting that we connected to these devices under controlled experimental settings, not in the wild. As a result, an external access control mechanism is necessary to protect such devices.

The above observations indicate that the address randomization, part of the BLE specifications, fails to provide the promised privacy protection. Various developers and manufacturers do not implement it properly; they rely on public Bluetooth addresses, apply weak randomization, or keep a consistent address for a long time. On the other hand, even if address randomization is properly implemented, other information in the advertisement or in the device might contain unique information (device name or id) that allows for its tracking.

Moreover, data accessed from an advertisement or the device (once connected) might reveal sensitive information about the user or the environment. Through our data collection campaign, we were able to detect different types of glucose monitors, wristbands, smart watches, fitness trackers, sleep monitors, laptops, smartphones, laptop security locks, security cameras, key trackers, headsets, etc. Knowing which type of glucose monitor the user is carrying or the type of physical security system s/he has installed could lead to serious harm to the user. Finally, an adversary might use such advertisement data as side information to infer more about the user’s behavior. For example, a temperature sensor constantly reading 60°F in winter would indicate a vacant property [41] which may invite in a thief.

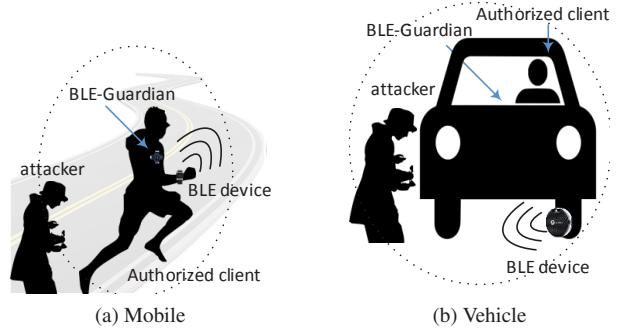


Figure 2: Example deployments of BLE-Guardian.

5 BLE-Guardian

BLE-Guardian addresses the aforementioned privacy threats by allowing only *authorized clients* to discover, scan, and connect to the user’s BLE-equipped device. Before delving into the inner workings of BLE-Guardian, we first describe the system and threat models.

5.1 System and Threat Models

5.1.1 System Model

A typical BLE scenario involves two interacting entities: the client and the BLE-equipped device. The BLE device broadcasts advertisements to make other nearby clients aware of its presence along with the services/information it is offering. A client can then connect to the device to access more services/information and control some of its attributes, in which case it will be the BLE-device’s gateway to the outside world.

The user’s mobile device (e.g., smartphone or tablet) acts a gateway where BLE devices are wearable (e.g., fitness trackers), or mHealth devices (e.g., Glucose monitor) (Fig. 2a). In a home environment, the user’s access point, PC, or even mobile device, serves as a gateway for BLE devices that include smart appliances, webcams, physical security systems, etc. Last but not least, a smart vehicle or the driver’s mobile device operate as gateways (Fig. 2b) for the different BLE-equipped sensors in the vehicle, such as tire pressure.² An interested reader could consult the work of Rouf *et al.* [32] for a discussion on the security and privacy risks of a wireless tire pressure sensor.

BLE-Guardian leverages the existence of gateways near the BLE-equipped devices to fend off unauthorized clients scanning and connecting to them. It comprises both hardware and software components. The hardware

²<https://my-fobo.com/Product/FOBOTIRE>

component is an external Bluetooth radio that connects physically to the gateway, while the software component is an accompanying application that runs on the gateway. BLE-Guardian requires another software component to run on the clients willing to discover and connect to the user’s BLE devices. The user, be it an owner of the BLE-equipped device or a client, interacts with BLE-Guardian through its software components.

5.1.2 Threat Model

BLE-Guardian only trusts the gateway on which it is running. Otherwise, the entire operation of BLE-Guardian will be compromised and will fail to provide the promised privacy provisions. BLE-Guardian achieves privacy protection at the device level, so that if it authorizes a client to access the BLE device, all applications running on that device will have same access privileges. This security/privacy dimension is orthogonal to BLE-Guardian and has been addressed elsewhere [28]. It also requires the user’s BLE device — the one to be protected — to comply with the BLE specifications.

BLE-Guardian protects a target BLE-equipped device against an adversary or an unauthorized/unwanted device that sniffs the device’s advertisements, issues scan requests and attempts to connect to the device. The adversary aims to achieve three objectives based on the BLE devices that the user is deploying:

1. **Profiling:** The adversary aims to obtain an inventory of the user’s devices. Based on this inventory, the adversary might learn the user’s health condition, preferences, habits, etc.
2. **Tracking:** The adversary aims to monitor the user’s devices to track him/her over time, especially by exploiting the consistent identifiers that the devices leak as we observed in Section 4.
3. **Harming:** The adversary aims to access the user’s BLE device to learn more sensitive information or even to control it. Both will have severe consequences for the user, especially if a certain device is known to have some vulnerabilities that allow remote unauthorized access [26].

This adversary can have varying passive and active capabilities, from curious individuals scanning nearby devices (e.g., using a mobile app), to those with moderate technical knowledge employing commercial sniffers, all the way to sophisticated adversaries with software-defined radios.

A *passive* attacker is capable of sniffing all the communications over advertisement channels including those that fail the CRC check. This includes all commercial Bluetooth devices and existing Bluetooth sniffers in the

market, such as the Texas Instruments CC2540 chip. The adversary might possess further capabilities by employing MIMO receiver that could recover the original signal from the jammed signal [38], especially in stationary scenarios. We refer to this adversary as the *strong passive* attacker.

Furthermore, the adversary is capable of injecting traffic into any Bluetooth channel at any given point of time, but will “play” within the bounds of the BLE specifications when attempting communication with the BLE device. This is a reasonable assumption, as the device won’t otherwise respond to any communication. We refer to such an adversary as the *active* attacker. On the other hand, the attacker might be able to transmit with higher power than allowed by regulatory bodies, in which case we refer to as the *strong active* attacker.

Thus, we have four classes of attackers referring to the combinations of their passive and active capabilities as shown in the first column of Table 4.

Attacks, including jamming the channel completely, masquerading as fake devices to trick the users to connect to them, or attacking the bonding process are orthogonal to our work. Such attacks have been addressed by O’Connor and Reeves [29] and Ryan [33]. Finally, once BLE-Guardian enables the authorized client to connect to the BLE device, it won’t have any control over what follows later.

5.2 High-Level Overview

BLE-Guardian is a system the user can use out of the box; it only requires installing a hardware component (an external Bluetooth radio) to the gateway and running an app on the gateway to control and interface with the Bluetooth radio. Conceptually, BLE-Guardian consists of *device hiding* and *access control* modules. The device hiding module ensures that the BLE device is invisible to scanners in the area, while the access control module ensures that only authorized clients are allowed to discover, scan, and connect to the BLE device.

Fig. 3 shows the high-level operation of BLE-Guardian from the moment a user designates a BLE device to be protected all the way to enabling authorized client connection to the protected device. The high-level operation of BLE-Guardian takes the following sequence:

1. The user installs the hardware component along with the accompanying app on the gateway.
2. The user runs the app, which scans for BLE devices nearby. The user can then choose a device to hide.
3. The device hiding module of BLE-Guardian starts by learning the advertisement pattern of the target BLE device along with that of the other devices in

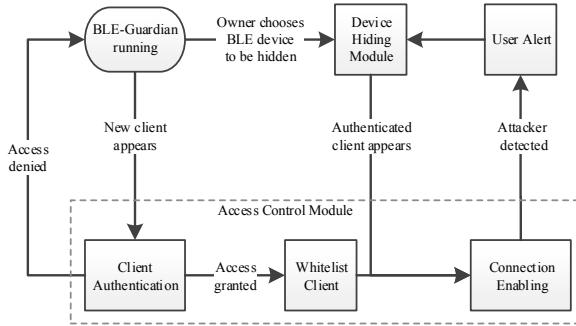


Figure 3: The modules of BLE-Guardian and their underlying interactions.

the area. The device hiding module then applies reactive jamming to hide the device.

4. When a new client enters the area and wants to discover the user's devices, it communicates with the access control module so that the user can either grant or reject authorization.
5. If the user authorizes the client, the access control module advertises *privately* on behalf of the BLE device to let the authorized client scan and connect to it.
6. BLE-Guardian monitors if other unauthorized entities are attempting to connect to the BLE device; in such a case, it blocks the connection and alerts the user.

5.3 Device Hiding

The hiding module is responsible for rendering the BLE device invisible to other scanning devices. The hiding module jams the device's advertisement session to achieve this invisibility. In particular, it targets three types of advertisements, ADV_IND, ADV_NONCONN_IND, and ADV_SCAN_IND of Table 1, which are periodic and leak more information about the user as we indicated earlier.

Hiding the BLE device is, however, challenging for two reasons. The hiding module must jam the BLE device precisely at the moment it is advertising. Also, it must not disrupt the operation of other devices advertising in the same area.

5.3.1 Learning

The hiding module first learns the target BLE device's advertising pattern before jamming to hide its presence. The device's advertisement pattern comprises the advertising interval, advertising channel sequence, and the time to listen on the individual channels. Fortunately, the

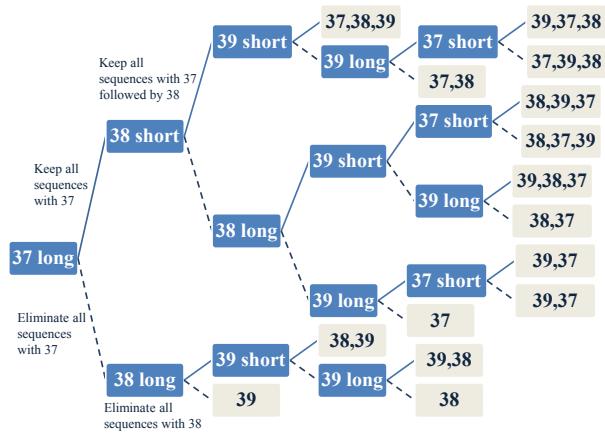


Figure 4: The learning algorithm followed by BLE-Guardian. The blue boxes refer to monitoring each channel either for a short period of time (less than 10ms) or for a longer period of 10.24 seconds. Depending on whether an advertisement is detected on the channel some sequences are eliminated till a sequence is decided on (gray boxes).

latter two parameters are deterministic and can be observed directly, which is not the case for the advertising interval. The BLE specification leaves it to the device to determine the advertising pattern, so that there are 15 possible permutations of the channel sequence.

As shown in Fig. 4, BLE-Guardian follows a process of elimination to identify the advertising sequence of the BLE device using a single antenna. In the worst case, it will take three advertising intervals to learn the entire advertising sequence of a BLE-equipped device. This corresponds to the longest path of Fig. 4, where BLE-Guardian monitors each channel for the maximum advertising interval of 10.24 seconds. At the same time, it would have identified the time the BLE device spends listening on each channel before switching to the next channel.

While observing the advertising sequence of the BLE device, the hiding module keeps track of the interval separating the consecutive advertisements sessions. The hiding module observes a set of inter-advertisement intervals, $t_i = adv + p$, where adv is the actual advertisement interval as set by the device and p is a random variable representing the random delay such that $p \in unif(0, 10ms)$. Also, BLE-Guardian will perform the same process for all advertising devices in the same area at the same time to learn their advertising parameters as well. Learning other devices' advertising at the same time will be useful as evident below.

5.3.2 Actuation

After identifying the advertising pattern, the hiding module needs to just detect the start of the advertisement session. Then, it jams the advertising channels according to their advertisement sequence. There is a caveat, though; the hiding module needs to detect the advertisement before it can be decoded. Otherwise, the rest of the jamming will not be effective.

From monitoring earlier advertisements, the hiding module obtains a set of t_i 's of different devices' advertisements, including the BLE device to be hidden. The advertisement interval will be $adv = t_i - p$ for each observed inter-advertisement interval. Each observed advertisement will be used to improve the estimation of the advertisement interval. For N observed intervals, we have:

$$adv = \frac{1}{N} \sum_{i=1}^N (t_i - p) = \frac{1}{N} \sum_{i=1}^N t_i - \frac{1}{N} \sum_{i=1}^N p. \quad (1)$$

Let $P = \frac{1}{N} \sum_{i=1}^N p$, the random variable P is drawn from the distribution $\frac{1}{N}p * \frac{1}{N}p * \frac{1}{N}p \dots \frac{1}{N}p$. Since the single random delays p are i.i.d., the mean of P will be equal to 5 (mean of the original distribution of p) and the standard deviation of $\sqrt{\sum_{i=1}^N \sigma_p^2} = \frac{5}{N\sqrt{3}}$. The hiding module estimates adv as:

$$adv' = E(adv) = \frac{1}{N} \sum_{i=1}^N t_i - 5. \quad (2)$$

The standard deviation of P will get lower as N increases; it defines the error in the estimate of adv as defined by Eq. (1). Given previous N observed advertisements from the BLE device, the hiding module can predict the next advertisement to happen at $adv_{next} \in [adv_{low}, adv_{high}]$ such that:

$$adv_{low} = T_N + adv' - e \quad (3)$$

$$adv_{high} = T_N + adv' + e + 10, \quad (4)$$

where T_N is the time of the last advertisement and e is the 90th percentile value of P (symmetric around the mean) which approaches 0 as N increases (so that $adv_{high} - adv_{low}$ approaches 10ms).

Starting from the last observed T_N of the target BLE device, the advertisement hiding module computes adv_{low} and adv_{high} . Also, it enumerates the list of other devices expected to advertise within the interval $[adv_{low}, adv_{high}]$.

The device hiding module always listens on the first channel of the advertising sequence of the BLE device to be hidden. During the interval $[adv_{low}, adv_{high}]$,

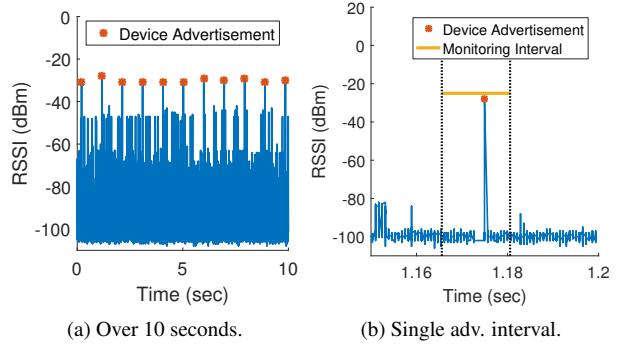


Figure 5: RSSI at channel 37 when a device is advertising at a distance of 1m at the interval of 960ms.

the device hiding module will sample the RSSI of the channel very frequently (every 25μs). When the received RSSI is $-90dBm$ or higher (the peaks of Fig. 5a), BLE-Guardian determines that there is a transmission currently starting to take place. The device hiding module moves immediately to jam the channel on which it is listening. Since the transmission of a typical advertisement message takes 380μs to finish [16], jamming the channel will prevent the message from being decoded by other receivers.

At this point, two situations might arise; (1) the target BLE device is the only device expected to be advertising at this time instant, or (2) some other device is expected to be advertising in the same interval. In the first situation, the target BLE device is most probably responsible for this transmission as part of its advertisement session. The device hiding module repeats the same process (sample RSSI and jam) over the rest of the channels to confirm that transmissions follow the device's advertising pattern. Fig. 5b shows an example interval where there is only one device advertising.

In the second situation, the device hiding module can't readily ascertain whether the transmission belongs to the target BLE device or not. This will take place when the observed transmission sequence matches the advertising sequence of the target BLE device and some other device that is expected to advertise at the same interval. To resolve this uncertainty, immediately after jamming the advertising message (400μs after commencing jamming on the channel), the device hiding module lifts jamming and sends scan requests for devices other than the target device. The device hiding module then listens on the channel to observe if a scan response is received. Despite its advertisement being jammed, any device will still be listening on and will respond to scan requests. Depending on whether a scan response is received or not, BLE-Guardian can associate the transmission with the correct device, be it the target BLE device or some other

device.

The device hiding module then adjusts the next monitoring interval according to the observed transmissions in the current intervals as follows:

$$adv_{low} = \min(T_N) + adv' - e \quad (5)$$

$$adv_{high} = \max(t_N) + adv' + e + 10, \quad (6)$$

where T_N represents the instants of the transmissions possibly matching the advertisement of the target BLE device in the current monitoring interval.

Note that we don't utilize the power level per se, or any physical-layer indicator, to indicate whether the same device is transmitting or not, as it is sensitive to the environment and the distance between BLE-Guardian and the target BLE device. To actually perform the jamming, the device hiding module continuously transmits at the maximum power for the specified interval.

BLE-Guardian may jam the advertisements of non-target devices which might disrupt their operation, which we referred to as the second situation above. Nevertheless, because of the random delay introduced by the device before each advertisement, the aforementioned "collision" events become unlikely. In Appendix A, we use renewal theory to show that the expected number of another device's advertisements within the expected advertising interval of the target BLE-equipped device will always be less than 1. This applies when BLE-Guardian protects a single BLE-equipped device. Our evaluation in Section 6 confirms this observation.

5.4 Access control

So far, BLE-Guardian has hidden the target BLE device, so neither authorized nor unauthorized entities have access to the device. It is the access control module that authorizes client devices and enables their access to the target BLE device.

5.4.1 Device Authorization

BLE-Guardian utilizes Bluetooth classic (BR/EDR) as an out-of-band (OOB) channel to authorize end-user devices intending to scan and access the target BLE device. BLE-Guardian runs in server mode on the gateway waiting for incoming connections, while the "authenticating" device will have BLE-Guardian running in client mode to initiate connections and ask for authorization. The choice of Bluetooth Classic as an OOB channel is natural; most end-user devices (such as smartphones) are dual-mode, supporting both BLE and Bluetooth classic. Moreover, Bluetooth classic already contains pairing and authentication procedures, eliminating the need for a dedicated authentication protocol. Last but not least, a

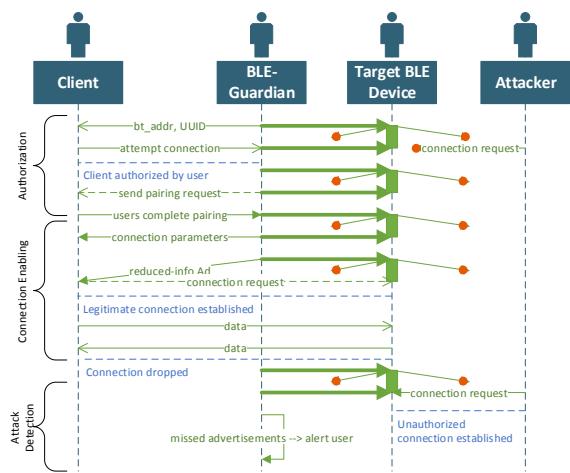


Figure 6: The sequence diagram of the access control module. Thin green lines from the target device designate the advertisements. Thick green lines from BLE-Guardian designate the jamming signal.

Bluetooth-equipped end-user device will be able to communicate simultaneously over Bluetooth classic and BLE so that it can communicate with both BLE-Guardian and the target BLE device.

Fig. 6 depicts the interactions between BLE-Guardian and a client device when they connect for the first time. BLE-Guardian will be listening on the gateway over a secure RFCOMM channel with a specified UUID. The gateway, however, won't be running in discoverable mode so as to prevent others from tracking the user. It is up to the party interested in authenticating itself to obtain the Bluetooth address of the user's gateway as well as the UUID of the authentication service.

Once the client end-user device obtains the Bluetooth address and UUID, it can initiate a secure connection to the gateway. This will trigger a pairing process to take place if both devices are not already paired. BLE-Guardian relies on Bluetooth pairing process to secure the connections between the gateway and the client device. For future sessions, an already paired client device can connect to BLE-Guardian without the need for any user involvement. The owner can also revoke the privileges of any client device by simply un-pairing it.

5.4.2 Connection Enabling

The device hiding module of BLE-Guardian jams the entire advertising sequence of the target BLE device, including the period it listens for incoming scan or connection requests so that it cannot decode them. Therefore, both unauthorized and authorized clients cannot access the target BLE device (the case of an adversary using

high transmission power will be discussed later). Fig. 6 shows the procedure that BLE-Guardian follows to enable *only* the authorized clients access to the target BLE device.

Immediately after the last advertisement of a single advertisement session, when the target device is the only one expected to be advertising, the access control module lifts the jamming. This ensures that the BLE device will not be advertising until the next advertising session, and it is currently listening for scan and connection requests. Then, BLE-Guardian advertises on behalf of the target BLE device on the same channel. The advertisement message contains only the headers and the address of the previously hidden device. It is stripped of explicit identifiers, hence leaking only limited information about the BLE device for a brief period.

At the same time, BLE-Guardian will communicate to the authenticated client app the address of the BLE device and a *secret* set of connection parameters over the OOB channel. BLE-Guardian’s app running on the client device will use the address and the parameters to initiate a connection to the BLE device. The connection initiation procedure is handled by the Bluetooth radio of the client device, which scans for the advertisement with the provided address. After receiving such an advertisement, it sends a connection request after which both devices will be connected.

The above procedure will not break the way BLE scans and connections take place. It doesn’t matter from which radio the actual advertisement was coming. From the perspective of the BLE device, it will receive a scan or connection request while waiting for one. On the other hand, the client device will receive an advertisement message while also expecting one.

5.5 Security and Privacy Features

BLE-Guardian addresses the tracking and profiling threats discussed in Section 5.1.2. It hides the advertisements, which are used as the main means to track users. It only exposes the advertisement for a very short period when enabling others to connect. Furthermore, BLE-Guardian greatly reduces the profiling threat by hiding the contents of the advertisement which leak the device name, type, and other attributes.

A strong passive attacker [38] can still detect the “hidden” peripheral by recovering the real advertisement, so that it can connect to the BLE-equipped device. Distinguishing legitimate connection requests based on the Bluetooth address of the initiator is not effective; an attacker could easily spoof its Bluetooth address to impersonate the authorized client. Therefore, BLE-Guardian uses the connection parameters of the connection request to distinguish fraudulent connection requests from legit-

Table 4: The protections offered by BLE-Guardian.

Adversary	Profiling Protect.	Tracking Protect.	Access Control	User Alert
Passive & Active	✓	✓	✓	✓
Strong Passive & Active	–	✓	✓	✓
Passive & Strong Active	✓	✓	–	✓
Strong Passive & Strong Active	–	✓	–	✓

imate ones. Legitimate connection requests contain the set of “secret” connection parameters communicated earlier to the client.

The probability of the attacker matching a particular set of connection parameters is very low. According to the specification, there are more than 3 million possible combinations of values for the connection, slave, and timeout intervals. If the connection is established based on a fraudulent connection request, then BLE-Guardian prevents the connection from taking place. The connection request already contains the hopping sequence initiation. BLE-Guardian hops to the next channel and jams it so as to prevent the BLE device from receiving any message from the connected unauthorized device. The BLE device drops the connection since it receives no message on the channel.

An attacker might abuse this connection process by constantly attempting to connect to the BLE device, thus depriving the authorized client of access. This will always be possible, even when BLE-Guardian is not deployed. BLE-Guardian observes such a situation from a high frequency of fraudulent connection requests and alerts the user of this threat. As it will be evident in Section 6, an active attacker injecting messages to the advertising channel can’t affect the operations of BLE-Guardian.

A strong active adversary, however, can override BLE-Guardian’s jamming and issue connection requests that the BLE-equipped device will decode. While jamming, BLE-Guardian runs in transmit mode and can not monitor the channel for incoming requests. Nevertheless, it detects that the BLE device is missing its advertising intervals, signifying that it was connected without BLE-Guardian’s approval. In such a case, BLE-Guardian alerts the user of the existence of a strong adversary nearby.

Finally, Table 4 summarizes BLE-Guardian’s capabilities when faced with the various adversaries described in Section 5.1.2.

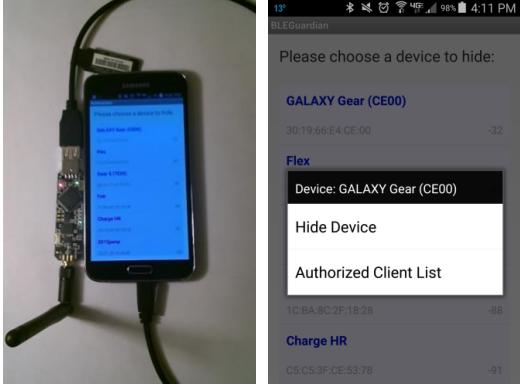


Figure 7: The deployment scenario for BLE-Guardian for a mobile user (left) and the main UI (right).

6 Implementation and Evaluation

We now present a prototype of BLE-Guardian along with its evaluation.

6.1 Implementation

We implement BLE-Guardian using Ubertooth One radio which is an open platform for Bluetooth research and development. It can connect to any host that supports USB such as Raspberry Pi, Samsung’s Artik-10, PC, smartphone (Fig. 7 (left)), etc. Since communication over USB incurs latency in the order of a few milliseconds, we implement most of BLE-Guardian’s functionalities inside Ubertooth One’s firmware to maintain real-time operation.

We also implement the software component of BLE-Guardian on Linux and Android. Fig. 7 (right) shows a screenshot of the BLE-Guardian app while running on Android in server mode where the user can choose the device to protect and control its authorized client list. The app communicates the Bluetooth address of the chosen device to the Ubertooth One radio.

BLE-Guardian requires running in privileged mode on the client device in order to be able to connect with modified connection parameters. This is easily achievable on Linux-based clients, but might not be the case for mobile devices. In other words, BLE-Guardian, while running in client mode on Android, requires root access to be able to issue connection requests with a set of specified connection requests. Also, BLE-Guardian (if running in privileged mode on the client device) can modify content of the advertisement message from the BLE scanner to the user-level application to reconstruct the original hidden advertisement. As such, user-level applications (on the trusted client) will receive the original advertisement, which does not break their functionality.

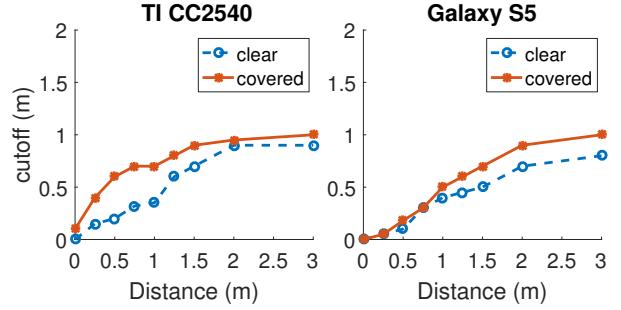


Figure 8: The cutoff distance as a function of the distance between BLE-Guardian and the target device.

Maintaining BLE-Guardian is easy; it only requires updating the application running on the gateway which usually takes place without the user’s intervention (e.g., mobile app updates). This application interacts with the hardware component and applies updates, if necessary, through pushing firmware updates, a process which is also transparent to the users.

6.2 Evaluation

To evaluate BLE-Guardian, we utilize Broadcom BCM20702A0 and Nordic nRF51822 chips as the target BLE devices (both transmitting at 4dBm) and the TI CC2540 dongle as the sniffer node. CC2540 is able to decode the messages on the three advertisement channels, even on those that fail the CRC check. We evaluate using Nordic and Broadcom chips instead of actual BLE products, because these products (such as Fitbits) are mostly powered by the same (Nordic and Broadcom) BLE chips.

6.2.1 Impact of Distance

Due to transmission power limitations (battery or regulatory bodies’ constraints), there would always be a small area around the target BLE device where BLE-Guardian won’t be able to enact the privacy protection. The transmission from the target BLE device covers the jamming signal of BLE-Guardian. Nevertheless, as the sniffer moves farther away from the target BLE device (in any direction), the jamming signal will cover the advertisements, provided that the BLE device and BLE-Guardian are not very far apart. So, there is a cutoff distance beyond which the adversary can’t scan, and connect to the target BLE device.

We study the cutoff distance of a target BLE device (advertising at 20ms) at different distances separating it from BLE-Guardian (between 0 and 3m). At each position, we move the sniffer node (either a CC2540 dongle or Samsung Galaxy S5) around the BLE device, and

record the farthest distance at which it received any advertisement from the BLE device as to study the hidden terminal effect. Furthermore, we repeat each experiment twice, the first with BLE-Guardian clear of any obstacles and the second with it inside a backpack.

It is evident from Fig. 8 that the cutoff distance increases as BLE-Guardian and the BLE device become farther apart. In all of the cases, however, the cutoff distance is less than 1m, even when BLE-Guardian and the BLE device are 3m apart. This also applies when BLE-Guardian is inside the backpack which should reduce the effectiveness of its jamming. Sniffing with a smartphone has a shorter cutoff distance because the smartphone’s BLE chip filters out advertisements failing the CRC check so that they are not reported to the OS.

The cutoff distance is enough to thwart tracking and profiling in several scenarios, especially when the user is moving (walking, jogging, biking or driving). In these scenarios, BLE-Guardian is not farther than 1m from the target BLE device. An adversary has to get very close to the user, even if BLE-Guardian is covered in a coat or bag, to be able to scan or connect to the BLE device.

In other cases, the user has to keep his BLE devices (to be protected) close to BLE-Guardian in order to get the best privacy protection possible. Our experiments showed that BLE-Guardian and the target BLE device must be separated by a maximum distance of 5m so that an attacker beyond the cutoff distance won’t be able to decode the advertisements. If BLE-Guardian and target BLE device are farther apart than this, then BLE-Guardian’s jamming won’t be able to cover the entire transmission area of the BLE device. In all circumstances, however, BLE-Guardian detects unauthorized connections and alerts the user accordingly.

6.2.2 Evaluation Setup

Beyond the cutoff distance, BLE-Guardian is capable of hiding the advertisements and controlling access to any target BLE device regardless of its advertising frequency. This protection, however, comes at a cost. In what follows, we evaluate BLE-Guardian’s impact on other innocuous devices, the advertising channel, and the gateway. In the evaluation scenarios, we deploy the target BLE devices at distance of 1.5m from BLE-Guardian, and the sniffer between BLE-Guardian and the BLE devices (at a distance of 0.5m from BLE-Guardian). We evaluate BLE-Guardian when protecting up to 10 target devices with the following advertising intervals: 10.24 sec (highest possible), 5 sec, 2.5 sec, 1.25 sec, 960ms, 625ms, 312.5ms, 100ms, 40ms, and 20ms (lowest possible). Note that evaluating with 10 target devices constitutes an extreme scenario; according to our dataset, the average user is bonded to less than 4 devices, which

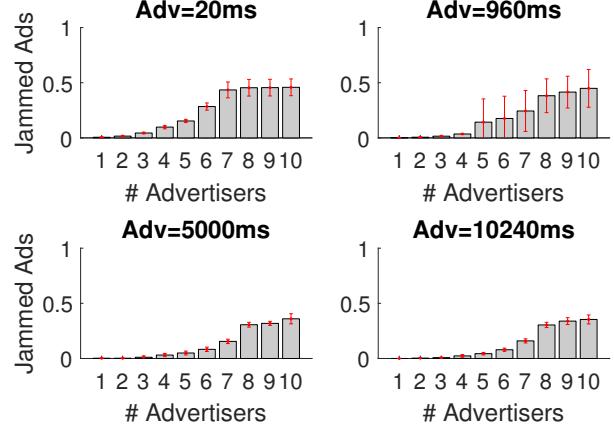


Figure 9: Portion of jammed advertisements of an innocuous BLE device when BLE-Guardian is running and protecting up to 10 advertisers.

would indicate the number of target devices (i.e. those to be protected).

6.2.3 Advertisement Hiding

Impact on Other Devices: We first evaluate the number of advertisements, not belonging to the target BLE device(s), BLE-Guardian will jam (Fig. 9). While accidentally jamming other devices doesn’t affect the privacy properties of BLE-Guardian, it hinders the services they offer to other users. In particular, we study four scenarios with an innocuous (not the target) BLE device advertising at 20ms, 960ms, 5s, and 10.24s, and a varying number (between 1 and 10) of target devices, which BLE-Guardian protects. Each subset of target devices of size N (≤ 10) contains the top N advertising intervals from the list of Section 6.2.2.

There are two takeaways from Fig. 9. First, BLE-Guardian has little effect on other devices when it protects a relatively low number of devices, or when the advertising interval of the target BLE device(s) is larger than 500ms; in these cases, BLE-Guardian will be less active (bars corresponding to less than 6 target devices in the four plots of Fig. 9). Second, BLE-Guardian has a higher effect on the innocuous device with higher advertising frequencies as observed from top-left plot of Fig. 9, especially when protecting a large number of devices (including those with 20 ms advertising interval).

In the latter case, BLE-Guardian is active for at least half of the time, representing the worst-case scenario of BLE-Guardian’s overhead where up to 50% of other devices’ advertisements are jammed. However, since the advertisement frequency is high, even with a relatively high rate of jammed advertisements, the user’s experience won’t be drastically affected. On the other hand,

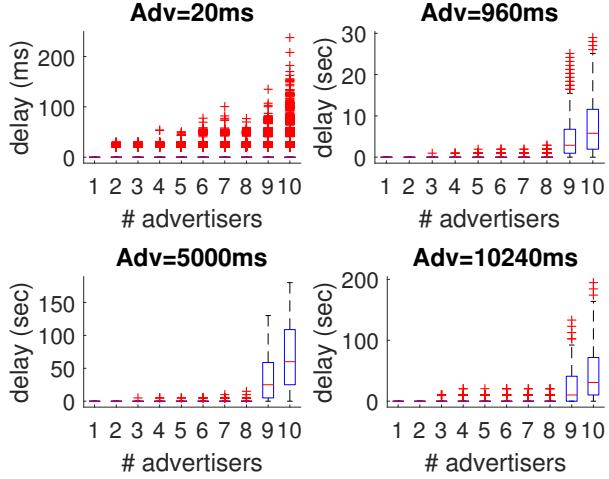


Figure 10: The delay of an authorized client in successfully connecting to the target device when BLE-Guardian is running.

when the target BLE device advertises at lower frequencies, the effect on the advertising channels and consequently other devices will be limited as evident from the rest of the plots of Fig. 9.

Impact on Authorized Access: To enable authorized connections, BLE-Guardian advertises on the behalf of the target BLE device only when it is confident that the target device is listening for connections. BLE-Guardian skips some advertising sessions which will introduce delays to authorized clients attempting connections as reported in Fig. 10. In this scenario, an authorized client is attempting connection to a target device advertising at 20ms, 960 ms, 5s, and 10.24 s, with an additional number of protected devices varying from 1 to 10. In the majority of the cases, the client has to wait for less than a second before successfully receiving an advertisement and issuing a connection. The only exception is the worst case consisting of BLE-Guardian protecting all of the 10 target devices (including devices advertising at intervals less than 100ms). The client might have to wait for up to multiple advertisement intervals before being able to connect. This is evident from the rightmost bar in each of the four plots of Fig. 10.

Impact on Advertising Channels Last but not least, we evaluate BLE-Guardian’s impact on the advertising channel, which, if high, might leak information about the existence of sensitive device(s). In this experiment, BLE-Guardian protects a single target device advertising at 20ms (the lowest possible), 960ms, and 10240ms (the highest possible). At the same time, two innocuous devices advertise at 20ms, in addition to other 15 devices not under our control advertising at different frequencies (minimum advertisement interval 30ms). In this sce-

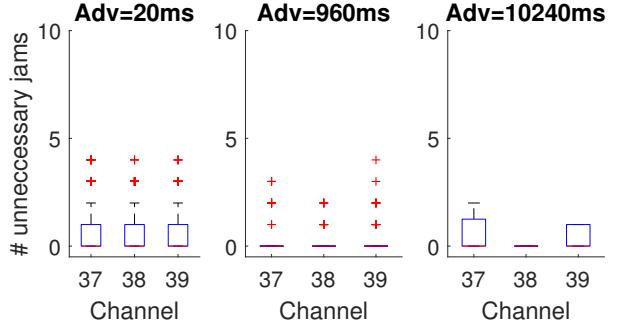


Figure 11: Unnecessary jamming instances with two advertisers at 20ms.

nario, BLE-Guardian will be active all the time since the two innocuous advertisers will force it to enlarge its monitoring interval between 20–30ms (while the advertising interval of the target device is only 20ms).

Fig. 11 shows the distribution of the number of unnecessary jammed instances in each interval when the target BLE device is expected to advertise. It is evident that in more than 50% of the intervals when BLE-Guardian is active, the number of unnecessary jamming instances events is 0, indicating a low overhead on the channel. When the target BLE device advertises at a lower frequency, BLE-Guardian is less active (middle and left plots of Fig. 11). These plots match the real-world scenarios observed from our data-collection campaign. Most commercial BLE devices advertise at relatively low frequencies (at intervals between 1 and 10s).

Finally, we evaluate the accuracy of predicting the next advertisement event of the target BLE devices. In all the experiments (including all scenarios), BLE-Guardian can predict the device’s advertisements, i.e., the target BLE device advertised in the interval it is expected to. BLE-Guardian is also able to jam all the advertisements of the BLE device over the three advertising channels. This indicates that an attacker can’t modify the behavior of BLE-Guardian by injecting traffic into the advertising channels.

6.2.4 Energy Overhead

BLE-Guardian incurs no energy overhead for both the target BLE devices and the authorized clients. Nevertheless, energy overhead is a concern when BLE-Guardian is attached to a smartphone. We measured the energy overhead of BLE-Guardian using a Monsoon power monitor while running on a Samsung Galaxy S4 with Android 4.4.2. In the idle case, BLE-Guardian consumes 1370mW on average. The average power consumption rises to 1860mW while transmitting and 1654mW while receiving as shown in Fig. 12a. Fortunately, BLE-Guardian doesn’t sense the channel or per-

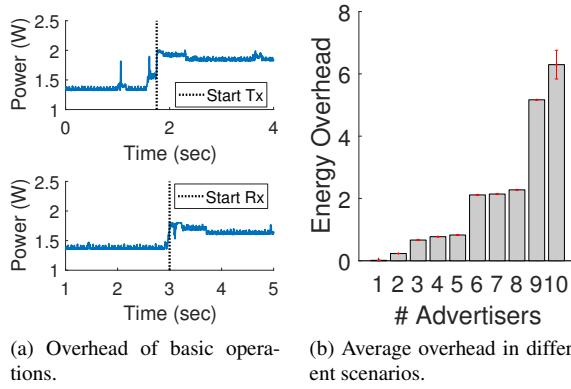


Figure 12: The energy overhead of BLE-Guardian running on Samsung Galaxy S4.

form jamming frequently. Fig. 12b shows the average energy overhead when BLE-Guardian is protecting the set of ten devices (we described earlier) at different advertising intervals. In the worst case of 10 target BLE devices, including a couple advertising at the highest frequency possible, the energy overhead is limited to 16% regardless of whether there are other advertisers in the area. In other cases, when there are less target devices and/or target devices are advertising at a lower frequency, the energy overhead is negligible.

7 Conclusion

BLE is emerging as the most prominent and promising communication protocol between different IoT devices. It, however, accompanies a set of privacy risks. An adversary can track, profile, and even harm the user through BLE-equipped devices that constantly advertise their presence. Existing solutions are impractical as they require modifications to the BLE-equipped devices, thereby making their deployment difficult. In this paper, we presented a device-agnostic system, called BLE-Guardian, which addresses the users' privacy risks brought by BLE-equipped devices. BLE-Guardian doesn't require any modification to the protocol and can be implemented with off-the-shelf Bluetooth hardware. We implemented BLE-Guardian using Ubertooth One radio and Android, and evaluated its effectiveness in protecting the users' privacy. In future, we plan to explore the data plane by analyzing and reducing the data leaks from BLE devices to unauthorized clients.

8 Acknowledgments

We would like to thank the anonymous reviewers for constructive comments. We would also like to thank Kr-

ishna C. Garikipati for useful discussions on this paper. The work reported in this paper was supported in part by the NSF under grants CNS-1114837 and CNS-1505785, and the ARO under grant W911NF-15-1-0511.

References

- [1] ARTICLE 29 DATA PROTECTION WORKING PARTY. Opinion 8/2014 on the recent developments on the internet of things. http://ec.europa.eu/justice/data-protection/article-29/documentation/opinion-recommendation/files/2014/wp223_en.pdf, Sep. 2014. Accessed: 18-01-2016.
- [2] ARUBA NETWORKS. Data Sheet: Aruba 320 series access points. http://www.arubanetworks.com/assets/ds/DS_AP320Series.pdf.
- [3] BLUETOOTH SIG. Bluetooth SIG Analyst Digest 2H 2014. <https://www.bluetooth.org/en-us/Documents/Analyst2014>. Accessed: 10-02-2016.
- [4] BLUETOOTH SIG. Specification of the Bluetooth System. Version 4.2, Dec. 2014. <https://www.bluetooth.org/en-us/specification/adopted-specifications>.
- [5] COX, D. *Renewal theory*. Methuen's monographs on applied probability and statistics. Methuen, 1962.
- [6] CRIST, R. Samsung swings for the fences with a new smart fridge at ces. <http://www.cnet.com/products/samsung-family-hub-refrigerator>, Jan. 2016. Accessed: 18-01-2016.
- [7] DAS, A. K., PATHAK, P. H., CHUAH, C.-N., AND MOHAPATRA, P. Uncovering privacy leakage in ble network traffic of wearable fitness trackers. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications* (New York, NY, USA, 2016), HotMobile '16, ACM, pp. 99–104.
- [8] DEGELER, A. Bluetooth low energy: Security issues and how to overcome them. <https://stanfy.com/blog/bluetooth-low-energy-security-issues-and-how-to-overcome-them/>, Jun. 2015. Accessed: 02-02-2016.
- [9] DIGI-KEY TECHNICAL CONTENT. Cypress PSoC 4 BLE (Bluetooth Low Energy). <http://www.digikey.com/en/articles/techzone/2015/dec/cypress-psoc-4-ble-bluetooth-low-energy>, Dec. 2015. Accessed: 12-01-2016.
- [10] FEDERAL BUREAU OF INVESTIGATION. Internet of Things Poses Opportunities for Cyber Crime. <https://www.ic3.gov/media/2015/150910.aspx>, Sep. 2015. Accessed: 18-01-2016.
- [11] FEDERAL TRADE COMMISSION. Internet of Things, Privacy & Security in a Connected World. <https://www.ftc.gov/system/files/documents/reports/federal-trade-commission-staff-report-november-2013-workshop-entitled-internet-things-privacy/150127iotrpt.pdf>, Jan. 2015.
- [12] GOLLAKOTA, S., HASSANIEH, H., RANSFORD, B., KATABI, D., AND FU, K. They can hear your heartbeats: Non-invasive security for implantable medical devices. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 2–13.
- [13] GREENSTEIN, B., MCCOY, D., PANG, J., KOHNO, T., SE-SHAN, S., AND WETHERALL, D. Improving wireless privacy with an identifier-free link layer protocol. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2008), MobiSys '08, ACM, pp. 40–53.

- [14] GRUTESER, M., AND GRUNWALD, D. Enhancing location privacy in wireless lan through disposable interface identifiers: A quantitative analysis. *Mobile Networks and Applications* 10, 3 (2005), 315–325.
- [15] HART, L. Telit Acquires Wireless Communications Assets to Boost Capabilities in Low-Power Internet of Things Market. <http://www.businesswire.com/news/home/20160113005310/en/>, Jan. 2016. Accessed: 01-02-2016.
- [16] HEYDON, R. *Bluetooth low energy: the developer's handbook*. Prentice Hall, 2012.
- [17] HILL, K. 'Baby Monitor Hack' Could Happen To 40,000 Other Foscam Users. <http://www.forbes.com/sites/kashmirhill/2013/08/27/baby-monitor-hack-could-happen-to-40000-other-foscam-users>, Aug. 2013. Accessed: 18-01-2016.
- [18] JIANG, T., WANG, H. J., AND HU, Y.-C. Preserving location privacy in wireless lans. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services* (New York, NY, USA, 2007), MobiSys '07, ACM, pp. 246–257.
- [19] JOHN PESCATORE. A SANS Analyst Survey: Securing the “Internet of Things” Survey. <https://www.sans.org/reading-room/whitepapers/analyst/securing-internet-things-survey-34785>, Jan. 2014. Accessed: 18-01-2016.
- [20] KUCHINSKAS, S. Bluetooth’s smart future in telematics. <http://analysis.tu-auto.com/infotainment/bluetooths-smart-future-telematics>, March 2013.
- [21] LEONARD, A. *Wearable Honeypot*. PhD thesis, Worcester Polytechnic Institute, 2015.
- [22] LESTER, S. The Emergence of Bluetooth Low Energy. <http://www.contextis.com/resources/blog/emergence-bluetooth-low-energy/>, May 2015.
- [23] LUTHRA, G. Embedded controllers for the Internet of Things. <http://www.edn.com/design/sensors/4440576/Embedded-controllers-for-the-Internet-of-Things/>, Oct 2015.
- [24] MADAAN, P. IoT for the smarter home. <http://www.ecnmag.com/article/2015/05/iot-smarter-home>, May. 2015. Accessed: 11-01-2016.
- [25] MARE, S., SORBER, J., SHIN, M., CORNELIUS, C., AND KOTZ, D. Hide-n-sense: Preserving privacy efficiently in wireless mhealth. *Mobile Networks and Applications* 19, 3 (2014), 331–344.
- [26] MARGARITELLI, S. Nike+ FuelBand SE BLE Protocol Reversed. <http://www.evilsocket.net/2015/01/29/nike-fuelband-se-ble-protocol-reversed/>, Jan 2015.
- [27] NANDUGUDI, A., MAITI, A., KI, T., BULUT, F., DEMIRBAS, M., KOSAR, T., QIAO, C., KO, S. Y., AND CHALLEN, G. PhoneLab: A large programmable smartphone testbed. In *Proceedings of SENSEMINES ’13* (New York, NY, USA, 2013), ACM, pp. 4:1–4:6.
- [28] NAVEED, M., ZHOU, X., DEMETRIOU, S., WANG, X., AND GUNTER, C. A. Inside job: Understanding and mitigating the threat of external device mis-bonding on android. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)* (2014), pp. 23–26.
- [29] OCONNOR, T., AND REEVES, D. Bluetooth network-based misuse detection. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (Dec 2008), pp. 377–391.
- [30] PARK, H., BASARAN, C., PARK, T., AND SON, S. H. Energy-efficient privacy protection for smart home environments using behavioral semantics. *Sensors* 14, 9 (2014), 16235.
- [31] PETERSON, A. Yes, terrorists could have hacked Dick Cheney's heart. <https://www.washingtonpost.com/news/the-switch/wp/2013/10/21/yes-terrorists-could-have-hacked-dick-cheneys-heart/>, Oct. 2013.
- [32] ROUF, I., MILLER, R., MUSTAFA, H., TAYLOR, T., OH, S., XU, W., GRUTESER, M., TRAPPE, W., AND SESKAR, I. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security’10, USENIX Association, pp. 21–21.
- [33] RYAN, M. Bluetooth: With low energy comes low security. In *Proceedings of the 7th USENIX Conference on Offensive Technologies* (Berkeley, CA, USA, 2013), WOOT’13, USENIX Association, pp. 4–4.
- [34] SCHNEIER, B. The internet of things is wildly insecure – and often unpatchable. <http://www.wired.com/2014/01/theres-no-good-way-to-patch-the-internet-of-things-and-thats-a-huge-problem>, Jan. 2014. Accessed: 18-01-2016.
- [35] SCHURGOT, M., SHINBERG, D., AND GREENWALD, L. Experiments with security and privacy in IoT networks. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2015 IEEE 16th International Symposium on* (June 2015), pp. 1–6.
- [36] SHEN, W., NING, P., HE, X., AND DAI, H. Ally friendly jamming: How to jam your enemy and maintain your own wireless connectivity at the same time. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 174–188.
- [37] SRINIVASAN, V., STANKOVIC, J., AND WHITEHOUSE, K. Protecting your daily in-home activity information from a wireless snooping attack. In *Proceedings of the 10th International Conference on Ubiquitous Computing* (New York, NY, USA, 2008), UbiComp ’08, ACM, pp. 202–211.
- [38] TIPPENHAUER, N., MALISA, L., RANGANATHAN, A., AND CAPKUN, S. On limitations of friendly jamming for confidentiality. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 160–173.
- [39] TURK, V. The internet of things has a language problem. <http://motherboard.vice.com/read/the-internet-of-things-has-a-language-problem>, Jul. 2014. Accessed: 03-02-2016.
- [40] WANG, P. Bluetooth low energy-privacy enhancement for advertisement.
- [41] WANT, R., SCHILIT, B., AND JENSON, S. Enabling the internet of things. *Computer* 48, 1 (Jan 2015), 28–35.
- [42] ZIEGELDORF, J. H., MORCHON, O. G., AND WEHRLE, K. Privacy in the internet of things: threats and challenges. *Security and Communication Networks* 7, 12 (2014), 2728–2742.

A Analysis of Device Hiding

BLE-Guardian may jam the advertisements of non-target devices which might disrupt their operation, which we refer to as the second situation in Section 5.3.2. Nevertheless, because of the random delay introduced by the device before each advertisement, the aforementioned “collision” events become unlikely. In what follows, we show that the expected number of another device’s advertisements within the expected advertising interval of the target BLE-equipped device will always be less than 1, when BLE-Guardian protects a single BLE-equipped device.

One could view the advertising process of a single BLE-equipped device as a renewal process [5], where each event corresponds to an advertising session. The inter-arrival times, X_i , are nothing but the inter-advertising intervals defined as i.i.d. random variables such that $X_i \sim \text{unif}(\text{adv}, \text{adv} + 10)$. The n^{th} advertisement time $T_n = \sum_{i=1}^n X_i$ has the distribution defined by the n -fold convolution of distribution of X_i . As n increases, the probability distribution of the n -th advertisement spreads over a larger time interval defined as $A = [\text{n.adv}, \text{n.(adv} + 10)]$.

The device hiding module attempts jamming at an interval of width 10ms, as specified before. If this jamming interval falls within the expected advertising interval of some other device, A , then the second situation of Section 5.3.2 might occur. Nevertheless, as n increases the length of interval A increases and thus the expected number of advertisements, from a single device within 10ms should be less than 1. We show below how the expected number of advertisements in a 10ms interval drops between $n = 1$ and $n = 2$. We consider $m(t)$, the expected number of events up to time t , defined as $F_X(t) + \int_0^t m(t-x)f_X(x)dx$, where $f_S(s)$ is the probability distribution of X_i which is equal to $\text{unif}(\text{adv}, \text{adv} + 10)$ and $F_X(t)$ is the cumulative distribution function given as:

$$F_X(t) = \begin{cases} 0 & t < \text{adv} \\ \frac{t-\text{adv}}{10} & \text{adv} \leq t \leq \text{adv} + 10 \\ 1 & t > \text{adv}. \end{cases} \quad (7)$$

During the first advertising interval, $t \in [\text{adv}, \text{adv} + 10]$, the expected number of advertisements is $\frac{\text{t-adv}}{10} +$

$\int_{\text{adv}}^t m(t-x)dx$ after substituting $F_X(t)$ and $f_X(x)$ with their corresponding expressions. After performing a substitution of variable of $y = t - x$, and since $m(t) = 0$ for $t < \text{adv}$, then $m(t) = \frac{t-\text{adv}}{10}$. So, if the expected advertising interval of the device hiding module overlaps with the first advertising interval of another advertising device, the expected number of events, $m(\text{adv} + 10) - m(\text{adv})$, will be 1, which is intuitive.

The second advertisement will take place at the interval $B = [2.\text{adv}, 2.(\text{adv} + 10)]$, and we use a similar procedure to derive the expressions for $m(t)$ for $t \in [2.\text{adv}, 2.\text{adv} + 10]$ and $t \in [2.\text{adv}, 2.(\text{adv} + 10)]$. If the expected advertising interval of the device hiding module overlaps with interval, B , then the expected number of advertisements $m(t+10) - m(t)$ will drop to $\frac{1}{2}$. The same trend will follow for the subsequent advertising intervals; the expected number of another device's advertisements within the expected advertising of the target BLE device will always be less than 1. Our evaluation in Section 6. confirms this observation.

Finally, even if another device, with the same advertising parameters, starts advertising with the target BLE device at the same time, their advertising events will eventually diverge. After N advertisements from both devices, the distribution of $Ta_{N+1} - Tb_{N+1}$, the difference in time between the $N + 1$ advertising instants of both devices will be a random variable with mean 0 but with $\sigma = 2.N.\frac{5}{\sqrt{3}}$. As N increases, the standard deviation increases, which in turn decreases the probability of both advertising events taking place within 10ms. The 10ms-advertising interval is the length of interval that the device hiding module expects the target BLE device to advertise.

Privacy in Epigenetics: Temporal Linkability of MicroRNA Expression Profiles

Michael Backes

backes@cispa.saarland

CISPA, Saarland University & MPI-SWS
Saarland Informatics Campus

Mathias Humbert

mathias.humbert@cispa.saarland

CISPA, Saarland University
Saarland Informatics Campus

Pascal Berrang

pascal.berrang@cispa.saarland

CISPA, Saarland University
Saarland Informatics Campus

Andreas Keller

andreas.keller@ccb.uni-saarland.de

Clinical Bioinformatics, Saarland University
Saarland Informatics Campus

Anne Hecksteden

a.hecksteden@mx.uni-saarland.de

Institute of Sports and Preventive Medicine
Saarland University

Tim Meyer

tim.meyer@mx.uni-saarland.de

Institute of Sports and Preventive Medicine
Saarland University

Abstract

The decreasing cost of molecular profiling tests, such as DNA sequencing, and the consequent increasing availability of biological data are revolutionizing medicine, but at the same time create novel privacy risks. The research community has already proposed a plethora of methods for protecting *genomic* data against these risks. However, the privacy risks stemming from *epigenetics*, which bridges the gap between the genome and our health characteristics, have been largely overlooked so far, even though epigenetic data such as microRNAs (miRNAs) are no less privacy sensitive. This lack of investigation is attributed to the common belief that the inherent temporal variability of miRNAs shields them from being tracked and linked over time.

In this paper, we show that, contrary to this belief, miRNA expression profiles can be successfully tracked over time, despite their variability. Specifically, we show that two blood-based miRNA expression profiles taken with a time difference of one week from the same person can be matched with a success rate of 90%. We furthermore observe that this success rate stays almost constant when the time difference is increased from one week to one year. In order to mitigate the linkability threat, we propose and thoroughly evaluate two countermeasures: (i) hiding a subset of disease-irrelevant miRNA expressions, and (ii) probabilistically sanitizing the miRNA expression profiles. Our experiments show that the second mechanism provides a better trade-off between privacy and disease-prediction accuracy.

1 Introduction

Since the first sequencing of the human genome in 2001, tens of thousands of genomes and over a million genotypes have been sequenced. The knowledge of our genetic background enables to better predict, and thus anticipate, the risk of developing several diseases, includ-

ing cancers, cardiovascular and neurodegenerative diseases. Moreover, the genomic research progress enables the development of personalized treatment through pharmacogenomics, studying the effect of the genome on drug response. One of the most important negative counterparts of this genomic revolution is the threat towards genomic privacy [11, 39]. Genomic data contains very sensitive information about individuals' predisposition to certain severe diseases, about kinship, and about ethnicity, all of which can lead to various sorts of discrimination. Furthermore, genomic data is very stable over time and correlated between family members [28]. Therefore, a lot of research has already been carried out to improve the genomic-privacy situation (most of the related literature is surveyed in [20, 42]).

However, our genome is not the only element influencing our health. Environmental factors (e.g., pollution, diet, lifestyle, ...) often play a crucial role in the development of most common diseases. Epigenetics (or epigenomics), transcriptomics, and proteomics aim to bridge the gap between the genome and our health status. Multi-omics research is a logical complementary step to genome sequencing: the DNA sequence tells us what the cell could possibly do, while the epigenome and transcriptome tell what it is actually doing at a given point in time. Using a computer analogy, if the genome is the hardware, then the epigenome is the software [16].

Despite the growing importance of epigenetics in the biomedical community, privacy concerns stemming from epigenetic data have received little to no attention so far. With the increasing understanding of epigenetics, it becomes clear that epigenetic data contains a vast amount of additional sensitive information, and can thus raise potential privacy risks. For example, a large number of severe diseases (such as cancers, diabetes, or Alzheimer's [21, 33, 46, 53]) are already identified to be affected by epigenetic changes and a recent study found that epigenetic alterations could even affect sexual orientation [43]. Furthermore, epigenetic data can potentially

tell us more about whether someone is carrying a disease at a given point in time, compared to the genome that only informs about the *risk* of getting certain diseases.¹ Moreover, it is still unclear whether the current genetic nondiscrimination laws would apply to epigenetic data. For instance, the US Genetic Information Nondiscrimination Act (GINA) is limited to genetic characteristics and epigenetic data might not be considered genetic information [18, 47].

In this work, we focus on microRNAs (abbreviated miRNAs), an important element of the epigenome discovered in the early 1990s. MiRNAs are small RNA molecules that regulate the majority of human genes. Studies of miRNA expression profiles have shown that dysregulation of miRNA is linked to neurodegenerative diseases, heart diseases, diabetes and the majority of cancers [21, 33, 40, 46, 53].² Therefore, miRNA expression profiling is a very promising technique that could enable more accurate, earlier and minimally invasive diagnosis of major severe diseases. As a consequence, it will certainly be increasingly used in medical practice.

In contrast to the DNA sequence, which mostly stays constant over time, it is believed in the biomedical community that the miRNA expression levels are varying sufficiently to invalidate any linkability attempts over time, thus naturally protecting personal privacy. This work, however, shows the contrary: despite their temporal variability, microRNA expression profiles are still identifiable and linkable after time periods of several months.

Contributions. In this paper, we study the temporal linkability of personal miRNA expression profiles, by presenting and thoroughly evaluating different attacks, and proposing defense mechanisms to enhance unlinkability.

Specifically, we first study an identification attack, which pinpoints a specific miRNA expression profile in a database of multiple expression profiles by knowing the targeted profile at another point in time. Second, we study a matching attack, which tracks a set of miRNA expression profiles over time. We rely on principal component analysis to pre-process the miRNA expression levels, and on a minimum weight assignment algorithm for the matching attack. We thoroughly evaluate these linkability attacks by using three different longitudinal datasets: (i) the blood-based miRNA expression levels of athletes at two time points separated by one week, (ii) the plasma-based miRNA expression levels of the same athletes at two time points separated by one week, and (iii) the plasma-based miRNA expression levels of patients with lung cancer over more than 18 months and eight time points. Our experimental results show that blood

miRNA expression profiles are about twice as easy to track over time compared to plasma miRNA profiles, and that the matching attack is more successful than the identification attack: We reach a success rate of 90% with blood and a success rate of 48% with plasma miRNAs in the matching attack whereas, in the identification attack, we reach a success rate of 76% with blood and 28% with plasma miRNAs. Moreover, we demonstrate that 10% of the miRNAs are already sufficient to achieve similar success rates as with all miRNAs. With the third dataset, we also observe that the attack achieves a similar success up to 12-month time periods.

We present two countermeasures to improve the unlinkability of miRNA expression profiles: (i) hiding a subset of the miRNA expressions, e.g., those that are not relevant for medical practice, and (ii) disclosing noisy miRNA expression profiles by adding noise in a differentially private and distributed manner. While the first countermeasure is useful especially in a clinical setting, in which the disease-relevant miRNAs are already known, the second countermeasure is intended to be better suited for the biomedical research community. In this context, as one of the objective is to discover associations between miRNAs and diseases, it is impossible to restrict the released data to only a few miRNAs.

We evaluate our protection mechanisms with the first aforementioned blood-based miRNA profiles of athletes and a fourth, also blood-based, miRNA dataset of more than 1,000 participants that includes information about 19 diseases (at a single point in time). The former is used to measure how temporal linkability is reduced with our countermeasures, whereas the latter helps us evaluate the evolution of accuracy (i.e., utility) in predicting patients' diseases from their miRNA expressions. The experiments show that it is possible to decrease linkability by at least 50% for almost no loss of accuracy (< 1%) for the majority of diseases with the noise mechanism. Moreover, our results demonstrate that the noise mechanism provides better privacy-utility trade-offs than the hiding method in 17 out of 19 of diseases, while allowing more flexibility in the data usage for biomedical researchers. This finding is reinforced by the fact that an adversary could use correlations between miRNA expressions to infer more miRNA expressions than those actually shared by our first countermeasure.

Organization. In Section 2, we present the biomedical background relevant to understand our work. In Section 3, we introduce the adversarial model. We then describe in detail our four datasets in Section 4. In Section 5, we present the analytical tools used to carry out our linkability attacks and our experimental results. In Section 6, we propose and evaluate countermeasures and compare their performance. We present the related literature in Section 7 before concluding in Section 8.

¹The only exception to this rule are Mendelian disorders, such as cystic fibrosis, which are largely determined by our genes.

²Known relations between miRNA and human pathologies can be found at <http://www.cuilab.cn/hmdd>.

2 Background

We briefly review the genetic concepts useful for understanding our paper. Epigenetics etymologically come from the combination of *epi*, which means “above”, “over” in Ancient Greek, and *genetics*, which means “origin”. This term broadly refers to the study of cellular and phenotypic trait variations stemming from other causes than changes in the genotype. These external factors are for example the in-utero or childhood development, environmental chemicals, aging or diet. Epigenetics can also refer to the changes themselves, such as DNA methylation and histone modification, which alter how genes are expressed without modifying the genome.

MicroRNAs (miRNAs) are epigenetically regulated mechanisms discovered in the early 1990s. MiRNAs are small non-coding RNA molecules that regulate gene expression in plants and animals. It has been shown that 60% of genes coding human proteins are regulated by miRNAs [25]. Whereas a miRNA is a RNA molecule containing around 22 nucleotides, *miRNA expression* is a real-valued number quantified in a two-step polymerase chain reaction (PCR) process. Different sets of miRNAs are expressed in different cell types and tissues.

Biomedical research is notably interested in discovering how miRNA expression affects physiological and pathological processes.³ Studies of miRNA expression profiling have demonstrated that dysregulation of miRNA is linked to neurodegenerative diseases (Alzheimer’s and Parkinson’s), heart diseases, diabetes, and the majority of cancers [21, 33, 40, 46, 53]. MiRNA expression profiling is hence a very promising technique that could enable more accurate, earlier and minimally invasive diagnosis of severe diseases. To mention one current, concrete application, miRNA expressions taken from *blood* samples suffice to detect several diseases, such as cancer or Alzheimer’s [34, 37]. In the following, we study the temporal linkability of miRNA expression profiles coming from blood and plasma (serum) samples.

3 Adversarial Model

We assume the adversary gets access to miRNA expression profiles of individuals at different points in time. Such epigenetic data is increasingly available in public research databases, such as the Gene Expression Omnibus (GEO) [4] or ArrayExpress [1] databases. Moreover, such data could be leaked through a major security breach, e.g., of a hospital server. Health data is

³Strictly speaking, miRNA is part of the epigenome while miRNA expression is generally considered more as part of the transcriptome. In this paper, we use the term epigenetics in its broad acceptance.

also increasingly available on the black market. For instance, cyber attacks against healthcare companies have increased by 72% from 2013 to 2014 [3]. Moreover, 91% of healthcare companies have experienced a violation of their databases over the last two years, and only 32% feel they have adequate resources to defeat these incidents [6]. Real-world cyber attacks show us that health data can be hacked en masse [5, 8] or that attacks can be more targeted towards high-profile victims [9]. Very sensitive medical data of thousands of patients can also end up online due to a human mistake [2].

In a typical scenario, the adversary would get access to miRNA expression levels of one or multiple individuals from a (private) health insurance or hospital database, and wants to match them with a (public) research dataset of miRNA expression levels at another point in time. A particularly sensitive scenario would be the matching of non-anonymized healthy miRNA samples with miRNA profiles that are known to be associated with diseases. Also note that researchers have demonstrated that RNA expression profiles could be matched to genotypes by relying on expression quantitative trait loci (eQTLs) [48]. Therefore, if the adversary can also access the genotypes of the victims, these genotypes provide him with further means for de-anonymizing the corresponding (micro)RNA expression profiles [26, 30]).

4 Dataset Description

Unlike in other fields of privacy research, where large amounts of data can be collected in a small amount of time and at low cost, in the health-privacy field we face the exact opposite: measuring the miRNA expression levels of one single sample already costs several hundred dollars. Longitudinal epigenetic data are particularly valuable, since patients have to regularly provide their biological samples over a long period of time. Therefore, the four datasets used throughout the paper, and described hereunder, represent very rich data.

We start by describing our three longitudinal datasets. The first dataset contains the blood-based miRNA expression levels of 29 well-trained male athletes (15 endurance athletes and 14 strength athletes) at two points in time, while the second dataset contains the plasma-based miRNA expression levels of those athletes at the same points in time.⁴ None of the athletes is known to be affected by a disease. The samples were taken prior and post exercising (time period of one week), similar to the data previously presented in [12]. The athletes followed a 6-day training with two training sessions a day, except at day 4 when only one session was scheduled.

⁴We selected blood and plasma since these two body fluids are likely candidates as source for biomarkers in future applications.

The tests were conducted at Saarland University (Germany) for the endurance athletes, and at Ruhr University Bochum (Germany) for the strength athletes.

In order to confirm our results, we make use of a third, independent dataset. This dataset contains the miRNA expression data of plasma of 26 lung-cancer patients (9 females and 17 males) over a period of more than 18 months [38], at eight time points: before surgery (tumor resection), two weeks after surgery (abbreviated A.S. in the graphs), and 3, 6, 9, 12, 15, and 18 months after surgery.⁵ The patients' ages range from 47 to 79. All three longitudinal datasets include the expression levels of 1,189 miRNAs for each individual at every time point.

Our last dataset contains the expression levels for 848 miRNAs collected from blood samples for each of 1,049 individuals [35] at only one time point. 94 of these individuals are considered to be healthy and are used as a control group in Section 6. Most of the rest represent cases, i.e., individuals carrying one out of the following 19 different diseases: 124 have Wilms tumor, 73 lung cancer, 65 prostate cancer, 62 myocardial infarction, 47 chronic obstructive pulmonary disease (COPD), 45 sarcoidosis, 45 ductal adenocarcinoma, 43 psoriasis, 37 pancreatitis, 35 benign prostate hyperplasia, 35 melanoma, 33 non-ischaemic systolic heart failure, 29 colon cancer, 24 ovarian cancer, 23 multiple sclerosis, 20 glioma, 20 renal cancer, 18 periodontitis, and 13 stomach tumor.

Note that a miRNA expression generally takes values between 0 (meaning the miRNA is not expressed at all) and tens of thousands. As we will mention later, we typically filter out miRNA whose median expressions among all individuals are smaller than 50, since these are non-expressed or not expressed enough to be significant.

While the last two datasets are both freely available in the GEO database (see accession number GSE68951 and GSE61741), the datasets consisting of athletes' miRNA expressions are not yet publicly available, but will be made available soon.⁶ We also discuss ethical considerations and how we handled these datasets in Appendix A.1.

5 Linkability Attacks

We study the extent of the linkability threat (as described in Section 3) by means of two attacks. First, we describe the mathematical principles behind our attacks, and then evaluate their success on our three longitudinal datasets.

⁵Note that for the last two points in time, we have the miRNA profiles of 25 and 22 patients, respectively.

⁶Please contact Andreas Keller for more information about the access to these datasets.

5.1 The Attacks

The first attack, called *identification attack*, refers to a scenario in which the adversary knows the miRNA expression profile of a targeted individual and aims at finding the corresponding miRNA expression profile in a database of n miRNA expression profiles, e.g., later in time. The second attack, called *matching attack*, refers to the case where the attacker has access to two databases of miRNA profiles collected at different points in time and wants to match their elements together.

For both our attacks, as there are more than 1000 known miRNAs with real-valued expression levels, we apply a pre-processing step using principal component analysis (PCA) with whitening. In particular, we apply the probabilistic PCA model proposed by Tipping and Bishop [49], which relies on singular value decomposition. This PCA step projects the high-dimensionality miRNA expression vectors to smaller-dimensionality uncorrelated components. The whitening step divides the resulting PCA components by the number of samples multiplied by the singular values in order to provide uncorrelated expression vectors of unit variance. We then make use of the Euclidean distance between the miRNA expression vectors projected on the first c principal components.

In the identification attack, we assume the adversary has had access to the miRNA profile $\bar{\mathbf{r}}_k^{t_1}$, vector containing the miRNA expressions of an individual k at time t_1 , and he wants to identify this individual in a database of n miRNA expression profiles $\{\bar{\mathbf{r}}_i^{t_2}\}_{i=1}^n$ collected at time $t_2 \neq t_1$. After having extracted the c principal components from the whole dataset by using PCA, the adversary ranks the n profiles (projected on the c components) $\{\bar{\mathbf{r}}_i^{t_2}\}_{i=1}^n$ by decreasing distance to the targeted miRNA profile (also projected on the c components) $\bar{\mathbf{r}}_k^{t_1}$ and picks the profile with minimum distance to the targeted profile. Formally, the adversary will select the profile $\bar{\mathbf{r}}_{i^*}^{t_2}$ where

$$i^* = \arg \min_i \|\bar{\mathbf{r}}_i^{t_2} - \bar{\mathbf{r}}_k^{t_1}\|_2.$$

In the matching attack, the adversary has access to two databases of miRNA expression profiles at two different time points t_1 and t_2 . We assume that the databases are of sizes n_1 and n_2 , both strictly greater than 1. First, if $n_1 = n_2 = n$, the adversary will assign one miRNA profile at time t_1 to exactly one profile at time t_2 . In this case, the best assignment σ^* is the one that minimizes the sum of the distances between every matched pair:

$$\sigma^* = \arg \min_{\sigma} \sum_{i=1}^n \|\bar{\mathbf{r}}_{\sigma(i)}^{t_2} - \bar{\mathbf{r}}_i^{t_1}\|_2.$$

This problem boils down to finding a perfect matching on a weighted bipartite graph, with n vertices on both

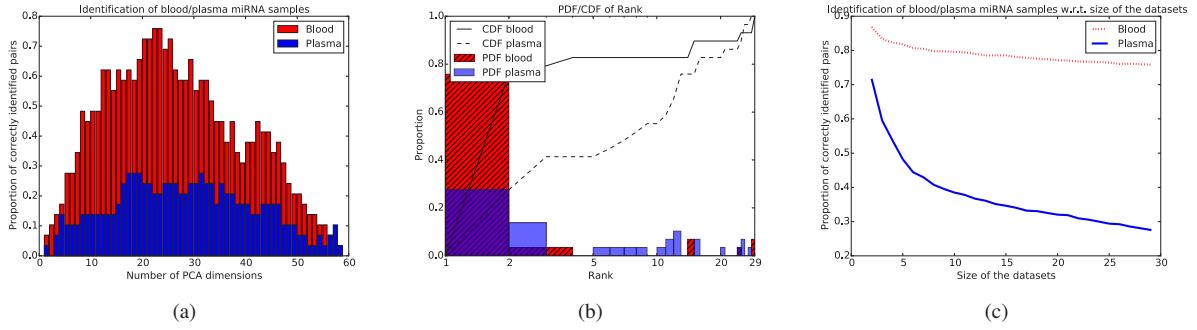


Figure 1: Success rate of the identification attack for the athletes dataset. (a) Proportion of successfully identified pairs plotted against the number of PCA dimensions (in $\{1, \dots, 58\}$). (b) Probability density function (PDF) and cumulative distribution function (CDF) of obtained ranks. (c) Proportion of successfully identified pairs plotted against the number of miRNA expression profiles.

sides representing the miRNA profiles, and a weight on each edge representing the Euclidean distance between any pair of miRNA profiles (vertices), projected on the first c principal components. We want to find the matching among $n!$ possible assignments that minimizes the sum of the weights between vertices. Fortunately, there exist several algorithms in the literature that find the minimum weight assignment in polynomial time. We use the blossom algorithm [19], because it only has a complexity of $O(n^3)$ and it can also be applied to general graphs.

If $n_1 \neq n_2$, we fill the smallest side of the bipartite graph with dummy vertices. Then we assign infinite weight to all edges from actual vertices to these dummy vertices in order to ensure that the dummy vertices will be the least likely assigned to the vertices in the largest side that are also present in the smallest side.

5.2 Experimental Results

We evaluate how successful both aforementioned attacks are in breaking the privacy of our three longitudinal datasets. We implement the attacks in Python, and make use of the libraries Scikit-learn [13, 44] (for PCA) and NetworkX⁷ (for the graph matching).

5.2.1 Identification Attack

In this subsection, we evaluate the success of an adversary, who aims at identifying the miRNA profile of a targeted individual in a longitudinal dataset. As mentioned in Section 4, the first two longitudinal datasets contain miRNA expression levels of 29 individuals collected at a time interval of one week.

First, we compare the success rate for correctly identifying samples for all possible PCA dimensions. Fig. 1(a)

indicates that the blood's miRNA expression levels are easier to identify over time than the plasma's miRNA expression levels. When identifying samples by their blood miRNA expression levels, we can reach a maximum success rate of 76% for the blood with 22 or 23 PCA dimensions. The maximum success rate for the plasma is 28% with 17, 18, 19 or 31 PCA dimensions. Note that both achieve their highest success with a number of PCA dimensions around 20.

Next, we rank the miRNA profiles at time t_2 in order of increasing distance to the targeted profile $\mathbf{r}_k^{t_1}$. Fig. 1(b) shows the rank of the correct sample $\mathbf{r}_k^{t_2}$ by using 22 PCA dimensions for the blood and 18 PCA dimensions for the plasma. The correct profile is ranked within the top 2 profiles in more than 40% of the cases for the plasma, whereas the correct sample is ranked within the top 2 samples in 80% of the cases for the blood.

In order to get an impression on the attack's performance on larger datasets, we also analyze the success of the identification attack with respect to the number of participants in the dataset, i.e., we vary the number of profiles among which the attacker has to identify the targeted miRNA profile, again using 22 PCA dimensions for the blood and 18 PCA dimensions for the plasma. Intuitively, when the number of miRNA samples increases, the success rate of the attacker should decrease. In this experiment, we adjust the number n of miRNA profiles between 2 and 29 and evaluate the attacker's success on a subset of our datasets. In particular, for each number of profiles n , we randomly choose 1000 different combinations (or fewer if necessary) of n out of 29 miRNA profiles and run the identification attack on every sample within this subset. Fig. 1(c) depicts the average success rates for each number of profiles n . As expected, the success rate monotonically decreases with the number of participants for blood and plasma samples. For plasma,

⁷<https://networkx.github.io>

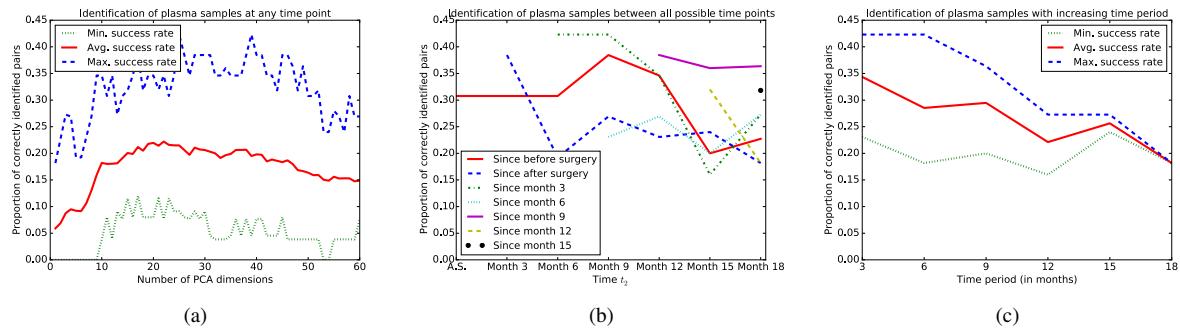


Figure 2: Success rate of the identification attack for the lung cancer dataset. (a) Success rate aggregated over all identifications between any t_1 and t_2 plotted against the number of PCA dimensions. (b) Success rate of identifying the miRNA profiles between time pairs t_1 and t_2 . (c) Success rate plotted against the time period between t_1 and t_2 .

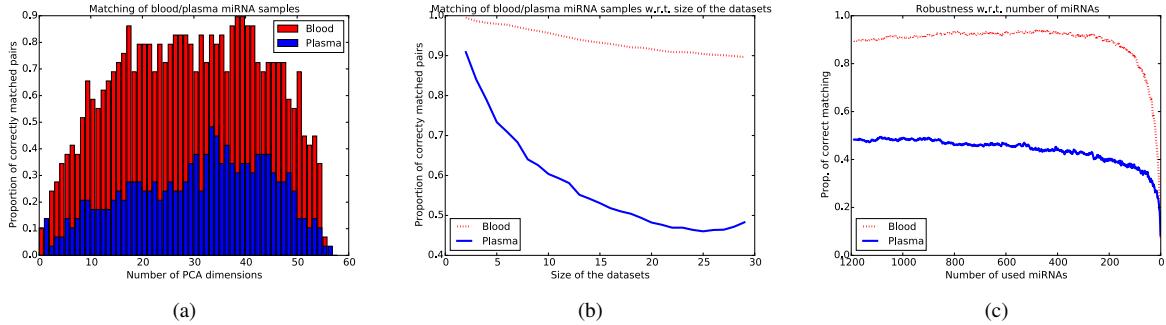


Figure 3: Success rate of the matching attack for the athletes dataset. (a) Proportion of successfully matched pairs plotted against the number of PCA dimensions. (b) Proportion of successfully matched pairs plotted against the number of miRNA profiles. (c) Proportion of successfully matched pairs plotted against the number of revealed miRNAs.

however, this decrease is much sharper, confirming that the blood's miRNA expression levels provide means for easier identification. From the curves' slopes, we can predict that, for larger datasets, blood based samples will still be subject to a relatively high identification success.

In order to validate our findings, we also evaluate our experiments on our other longitudinal, independent dataset containing plasma miRNA profiles from 26 individuals with lung cancer collected over up to eight different points in time.

First, we evaluate the attacker's success with respect to a varying number of PCA dimensions. Fig. 2(a) depicts the minimum, average and maximum success rate of an attacker when identifying the samples between different points in time, irrespective of the time period between them. The maximum success rate for the identification attack is 42% and is achieved for 25 and 39 PCA dimensions. The usage of 22 PCA dimensions yields the highest average success rate, of 22%. The highest minimal success rate in the dataset is achieved for 17 PCA dimensions (12%).

These results are similar to what we obtained in our experiments for the athletes dataset: The best results are achieved for a number of PCA dimensions around 20 in both datasets. The highest average success rate lies 6 points below the best success rate for the athletes dataset. This could be explained by longer time periods in this dataset. However, for some time periods, we can achieve one and a half the success rate of the first dataset. When comparing the top 10 miRNAs contributing to the first PCA dimension in this dataset and in the athletes' plasma dataset, we also find an overlap of 80% between these miRNAs. This indicates that approximately the same set of miRNAs can be used to differentiate plasma expression profiles between individuals in both datasets. Thus, we can conclude that, while miRNA expression levels are directly linked to health status, the health status only affects a subset of the miRNAs, which has only little effect on the temporal linking.

To further investigate the effect of different time periods on the attacker's success, we plot the maximum success rates between all possible, ascending combinations

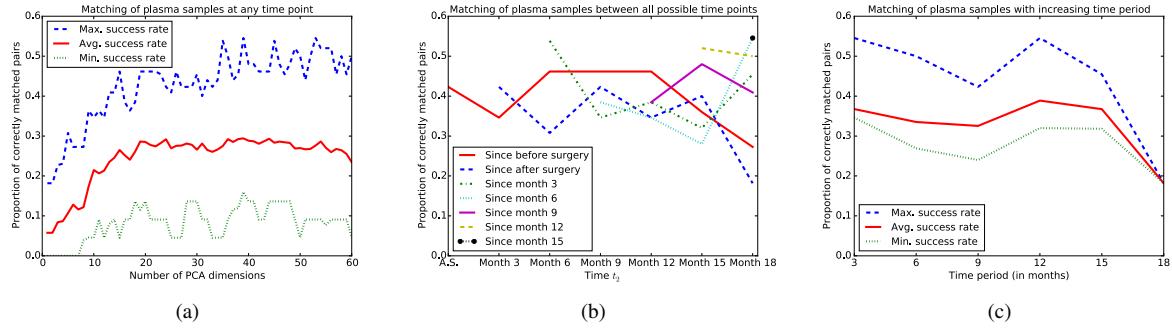


Figure 4: Success rate of the matching attack for the lung cancer dataset. (a) Success rate aggregated over all matchings between any t_1 and t_2 plotted against the number of PCA dimensions (in $\{1, \dots, 60\}$). (b) Success rate of matching the miRNA profiles between time pairs t_1 (various curves), t_2 (x-axis value). (c) Success rate plotted against the distance between t_1 and t_2 .

of points in time in Fig. 2(b). With only a few exceptions, the best success rates are most often achieved for consecutive time points. The only two exceptions are found for $t_1=\text{before the surgery}$ and $t_1=\text{the sixth month after the surgery}$. In general, however, we notice a tendency of slight decrease in success over an increasing time period.

In order to verify this finding, we group the results by the period between t_1 and t_2 (Fig. 2(c)). Note that, since we do not know the time period between before the surgery and after it, we leave out all results that use samples collected before the surgery. Clearly, the best achievable success rate drops for increasing time periods. This decrease over larger periods of time can partially explain the lower average success rate in this dataset compared to the athletes' dataset (considering a much smaller time period).

Next, we computed the guessing entropy [14, 41] for the identification attack. The guessing entropy $E[G(X)]$ is the expected number of guesses an adversary would need to identify the correct sample at a different point in time. For the identification attack it is given by $E[G(X)] = \sum_{i=1}^n i \cdot \Pr[X = i]$, where X denotes the rank of the correct sample at time t_2 and $\Pr[X = i]$ denotes the empirical probability that the correct sample is ranked at the i^{th} position.

For blood-based samples of our athletes dataset, the attack can achieve a guessing entropy just below 4, clearly outperforming random guesses, which would yield an entropy of 15 guesses on average. For plasma-based samples of the same dataset, the attack yields an entropy of approximately 9 guesses. This result is consistent with the results on the lung cancer dataset, where, on average, an adversary would need just fewer than 9 guesses (compared to a guessing entropy of 13.5 for random guesses). Moreover, for some t_1 and t_2 , the attack is even able to achieve a guessing entropy smaller than 6.

5.2.2 Matching Attack

We evaluate here the success of the adversary, who tries to link all participants over time, again for the three aforementioned longitudinal datasets. Starting with the athletes' datasets, we compare the success rate of matching the blood and the plasma over all possible PCA dimensions for 29 participants. In Fig. 3(a), we notice the same behavior as in the identification attack: the blood based miRNA expression levels are much easier to link over time than the plasma based levels. We even reach a higher maximum absolute success rate than in the identification attack: 90% with 39 or 40 PCA dimensions for the blood and 48% success with 34 PCA dimensions for the plasma samples.

The identification attack's lower success rate is due to the fact that it is evaluated for each sample individually, thus allowing multiple samples at t_1 to be linked to the same (potentially wrong) sample at t_2 . Since our perfect matching attack rules out those cases by forcing each profile at t_2 to be matched to exactly one profile from t_1 , it also decreases the number of wrongly matched samples.

Next, we also analyze the success of the attack with respect to the number of participants to be matched together. Intuitively, the more miRNA profiles there are, the more challenging it should be for the adversary to match them at different time points. Again, we make the number of participants n vary between 2 and 29 at both time points, again randomly sampling 1000 combinations (or fewer, if there are fewer than 1000 combinations) and averaging the result. Fig. 3(b) shows the expected trend of decreasing success for the blood miRNA samples. The plasma scenario monotonically decreases between 2 and 25 participants and then slightly increases until 29. This artifact could be explained by the smaller number of random combinations, and thus experiments, when $n > 26$. We also find that the blood attack faces

a rather linear decrease in success whereas the plasma success rate decreases much faster. By extrapolating this linear trend, we can expect a success rate as high as 60% with 120 participants in the datasets. Therefore, we again conclude that the blood has miRNA expression levels that enable much easier tracking over time than the plasma, which is consistent with the results of the identification attack.

Fig. 3(c) investigates how the attack’s success evolves when revealing only a subset of the miRNA expression levels. We gradually drop individual miRNAs in random order and compute the attack’s success. The figure shows the success rate (for each possible number $m \in \{1189, 1188, \dots, 2, 1\}$ of miRNAs) averaged over 50 randomly chosen orderings of miRNAs. We notice that the attack’s success is very stable, especially for the blood samples, from 1189 to 200 miRNAs. For the blood, the success decreases below 80% the first time when there are fewer than 100 miRNAs available to the adversary. We further study the implications of this robustness in the context of our countermeasures in Section 6.

We also made use of our third longitudinal dataset containing plasma miRNA expression profiles of 26 individuals over up to eight different time points (cf. Section 4). In Fig. 4(a), we see that the average success rate reaches its maximum at a number of PCA dimensions very close to the number of dimensions for the athletes dataset, i.e., 34. However, this maximum is approximately 30%, which is smaller than the 48% reached for the first dataset. A greater period between time points could explain this behavior, and we also see that we can still reach a maximum success rate of 55% between some time points, with 39 PCA dimensions. We explore the time effect in deeper details in the following figures.

Fig. 4(b) depicts the maximum success rate between any pair of time points t_1, t_2 . For instance, the solid red line shows the success rates between $t_1 = \text{before surgery}$ and all others. It is difficult to detect any trend with respect to the time period in the different curves, except a slight decrease when the time period is higher or equal to 15 months. This is confirmed by Fig. 4(c) that depicts the maximum, average, and minimum success rate with respect to the period between t_1 and t_2 . We clearly notice a decreasing rate between 3 and 9 months, an increase to 12 months, and finally clear decrease towards 15 and 18 months.

6 Countermeasures

In this section, we propose and evaluate two main defense mechanisms for preventing miRNA expression data from being tracked over time. The proposed techniques are based on well-established privacy-enhancing

methods, previously applied in other privacy contexts, such as location privacy. The first approach relies on a quite straightforward technique: release only a subset of the miRNAs. We can already see from Fig. 3(c) of Section 5 that the matching attack is quite robust to a decrease in the number of miRNAs. Nevertheless, we show hereafter how we can keep a high utility in combination with unlinkability of expression profiles over time by revealing a small subset of miRNAs. The second countermeasure consists in adding noise to the released miRNA expression vectors, independently for every individual. This method shows very promising results, reaching an even better privacy-utility trade-off than the hiding mechanism. Furthermore, we also investigate the effect of correlations between miRNA expression levels and present the privacy evolution when the adversary can infer missing miRNAs by using these correlations.

For evaluating the privacy provided by our defense mechanisms, we focus on the matching attack against blood-based miRNAs, as this constitutes the worst-case attack from a privacy perspective, as shown in Section 5. Moreover, we assume the attacker is able to select the number of PCA dimensions that maximizes his success. This provides us with a conservative measure of privacy, showing the worst-case privacy levels individuals can expect.

6.1 Baseline Utility

Before presenting the proposed countermeasures and their efficiencies, we must carefully define the context in which they should apply. Indeed, we can rarely have both perfect privacy and maximum utility, so that we often need a trade-off between these two. Therefore, the efficiency of the defense mechanism cannot only be judged based on the privacy metric, but must also relate to the utility brought in the context in which the data is used.

According to biomedical experts, miRNA expression profiles have strong potential to help predict various severe diseases, from cancer to Alzheimer’s disease. Biomedical researchers typically rely on standard machine learning algorithms to identify which miRNAs are playing a significant role in the disease of interest. They are dealing with binary classification, between cases (carrying the disease) and controls (healthy), and most often rely on support vector machines (SVMs). In particular, they typically use radial basis function SVMs and select a subset of features by subsequently adding miRNAs in order of their significance values (e.g., p -values computed by the Wilcoxon-Mann-Whitney (WMW) test) [37] or equivalently in order of their area under the ROC curve (AUC). Given samples of cases and controls, the accuracy is then defined as the number of correctly classified samples divided by the

Disease	Maximum accuracy with the best subset of expressed miRNAs (# miRNAs)	Accuracy with all expressed miRNAs
Periodontitis	0.941 (37)	0.88
Renal cancer	0.988 (32)	0.962
Wilms' tumor	0.95 (150)	0.937
Benign prostate hyperplasia	0.921 (105)	0.883
Chronic obstructive pulmonary disease	0.932 (70)	0.886
Colon cancer	1.0 (30)	0.997
Ductal carcinoma	0.938 (55)	0.92
Glioma	0.927 (19)	0.83
Lung cancer	0.899 (60)	0.848
Melanoma	0.996 (185)	0.992
Multiple sclerosis	0.992 (40)	0.979
Myocardial infarction	0.893 (400)	0.884
Nonischaemic systolic heart failure	0.9 (135)	0.871
Ovarian cancer	0.919 (18)	0.876
Pancreatitis	0.941 (130)	0.899
Prostate cancer	0.923 (90)	0.91
Psoriasis	0.914 (350)	0.902
Sarcoidosis	0.977 (200)	0.97
Tumor of stomach	0.969 (160)	0.89

Table 1: Accuracy of the SVM algorithm in classifying individuals between cases (carrying the disease) and controls (healthy), for 19 diseases, without countermeasure.

total number of samples. Note that we compute the average accuracy over a repeated k -fold cross-validation.

In this work, we define the utility as the accuracy of the SVM classifier, as defined above. We use a 10-fold cross-validation with 5 repeats (using R and the caret⁸ library) and determine the miRNAs' p -values by using the WMW test and adjusting the significance values for multiple tests using the Benjamini-Hochberg adjustment. The WMW test statistic is applied for each miRNA individually in order to test whether this miRNA has similar expressions between cases and controls (null hypothesis). The p -values then provide us with the relevance of the miRNA to the disease of interest. In contrast to the t -test, the WMW test can be applied on unknown distributions. This way, we follow the standard procedure of biomedical research. Table 1 shows the accuracy of

our SVM algorithm applied on our 1000+ participants dataset to predict 19 diseases, without any obfuscation. The maximum accuracy here is what we refer to as the baseline utility in the subsequent subsections.

Note that, before running the SVM algorithm, we filter out non-expressed miRNAs, i.e., those with a median level of expression smaller than 50 over the 1000+ individuals, which leaves us with 446 expressed miRNAs.

6.2 Hiding MicroRNA Expressions

The first countermeasure that we study is miRNA expression hiding. This obfuscation technique has the advantage to be non-perturbative, i.e., to preserve the correct values of all revealed miRNA expressions. However, as we have seen in Section 5, the attacks are extremely robust to removal of miRNAs. In the following, we want to find an optimal trade-off between the diagnosis accuracy, i.e., the utility, and the unlinkability of the data, i.e., the privacy. To this end, we make use of both our blood-based datasets, the 1000+ dataset with blood-based miRNA expressions to run our SVM algorithm and the athletes' dataset with blood-based miRNAs to evaluate the level of privacy. Note that we filter both datasets' miRNAs in order to have the same set of 446 miRNAs in both cases. While we measure the utility in terms of accuracy of the SVM, the privacy is measured in terms of the maximum achievable success rate (over all possible PCA dimensions) of our matching attack.

Figure 5 shows the evolution of privacy and utility for a range of 1 to 100 disclosed miRNAs, for 6 different severe diseases.⁹ We focus on this range of miRNAs as: (i) for more than 100 miRNAs, the attack's success rate is approximately the same as the one without countermeasure, and (ii) the SVM can already achieve very high accuracy with up to 100 miRNAs. We gradually reveal the miRNAs in decreasing order of significance (based on p -values), as computed in Subsection 6.1.

Figure 5 demonstrates that there exists a trade-off between the utility of miRNA expressions and the privacy of the contributors' data. Note that we also depict the relative decrease in accuracy compared to the maximum SVM accuracy computed in Subsection 6.1 and the relative decrease in the attack's success (increase in privacy) compared to the attack's success with all miRNAs, i.e., 90%. We see that the relative decrease in accuracy is almost always smaller than 10%. The only exceptions to this are with pancreatitis and melanoma, for fewer than 3 disclosed miRNAs. Moreover, regarding the privacy, the figures show that we can never reduce the attack's success by more than 50% when revealing more than 20

⁸caret.r-forge.r-project.org

⁹These are representative of the behavior of all 19 diseases we tested our privacy-preserving mechanisms on.

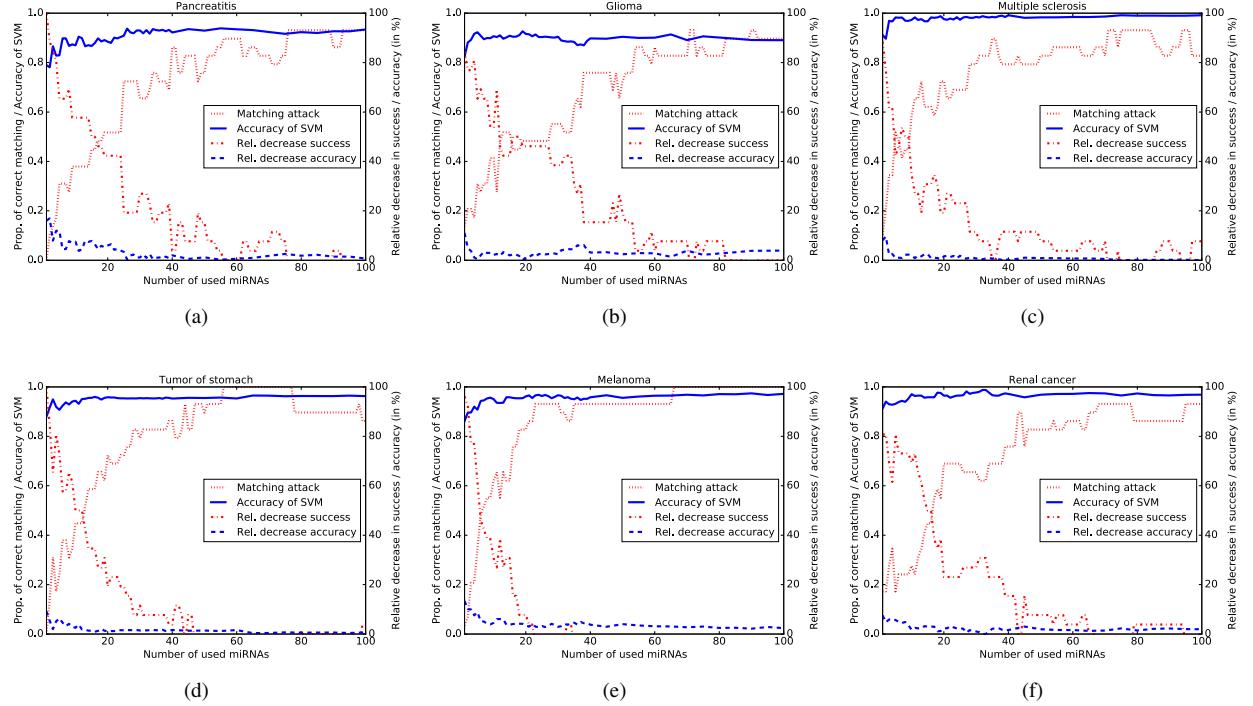


Figure 5: Evolution of privacy (unlinkability) and utility (classifier accuracy) plotted against the number of released miRNAs for the following diseases: (a) Pancreatitis, (b) Glioma, (c) Multiple sclerosis, (d) Tumor of stomach, (e) Melanoma, (f) Renal cancer. The *relative decrease success* curve refers to the decrease in success of the matching attack compared to the success without countermeasure. Similarly, the *relative decrease accuracy* curve refers to the decrease in accuracy of the SVM classifier with respect to the case without protection mechanism.

miRNAs. Nevertheless, within the range of 3 to 20 disclosed miRNAs, we can find, for all diseases, a satisfactory trade-off between utility and privacy.

In particular, for glioma, we can decrease the linkability attack’s success and thus improve the privacy by 80.8% when using 4 miRNAs, while reducing the classification accuracy by only 1.1%. Similarly for multiple sclerosis, 7 miRNAs provide an increase in privacy of 53.8%, while the decrease in accuracy only amounts to 0.9%. For renal cancer and 10 miRNAs, we are able to achieve an improvement in privacy of 69.2% and a decrease of accuracy of only 1.7%. There are only two diseases for which it is very difficult to have both unlinkability and very high utility: melanoma and pancreatitis. For melanoma, we notice that the matching attack’s success has a fast increase with very few miRNAs, and already exceeds 50% starting with only 7 miRNAs. For pancreatitis, the SVM’s accuracy is relatively low (compared to the maximum) for the first 20 miRNAs. Thus for both diseases, either privacy or utility would have to be slightly sacrificed for the other.

MiRNA co-expression. Like between variants in the genome, there exist correlations between miRNA expressions: around 40% of miRNAs are not independently expressed [7]. This means that the adversary, by knowing these correlations, could increase his knowledge about the non-disclosed miRNA expressions. In order to evaluate the importance of such correlations, we first compute the Pearson’s correlation coefficients, and their corresponding p -values, in all 99,235 pairs of the 446 expressed miRNAs in our fourth dataset. Filtering out all correlations with p -values greater than 0.001 (after Bonferroni correction for multiple correlations’ testing) or correlation coefficient smaller than 0.5 leaves us with 47% of miRNAs not independently expressed. Figure 6 shows the updates of the linkability attack’s success by taking into account all significant correlations as defined above. In our experiments, we take a quite conservative approach: We assume that the adversary can perfectly infer the miRNAs correlated with those that are gradually disclosed. The dotted curve provides an upper bound estimate on the success rate. A tighter bound could be derived by knowing more precisely the probabilistic dependencies between miRNAs. This is left for future work.

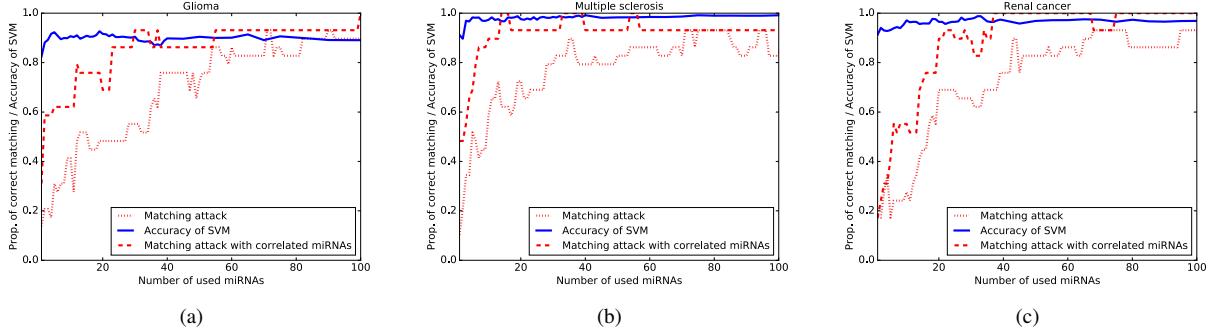


Figure 6: Correlations between miRNAs. Evolution of privacy and utility, when miRNAs correlated with the revealed miRNAs are taken into account for the attack. This provides an upper bound on the best linkability of miRNA expression profiles, i.e., worst-case privacy level. (a) Glioma, (b) Multiple sclerosis, (c) Renal cancer.

For Fig. 6, we make use of the three diseases of Figure 5 that gave best trade-off between privacy and utility, i.e., glioma, multiple sclerosis and renal cancer. We observe that the success rate knowing miRNAs correlated with disclosed miRNAs is much higher than without them, except for the very first miRNAs in Fig. 6(c). It shows that the most significant miRNAs for the SVM classification are co-expressed with others, which penalizes privacy significantly. Making use of the best subsets of miRNAs found above without correlations, containing 4 miRNAs for glioma, 7 for multiple sclerosis, and 10 for renal cancer, we evaluate the new privacy levels when miRNA correlations are taken into account. For glioma, instead of improving unlinkability by 80.8%, the 4 miRNAs and their correlated miRNAs yields an improvement in privacy of 34.6%. For renal cancers, the privacy enhancement drops from 69.2% to 38.5% and, for multiple sclerosis, knowing 7 miRNAs and their co-expressed miRNAs yield an attack’s success rate almost equal to the highest rate with the full set of miRNAs. However, we can find new, better trade-offs: e.g., disclosing 5 miRNAs for multiple sclerosis still provides the same high SVM accuracy (decrease of 0.9% compared to the baseline) while reducing the attack’s success by 23%. Note that we do not make use of the correlated miRNAs for the SVM algorithm as we are not certain about how they correlate with the disclosed ones.

6.3 Noise Mechanism

As we have noticed in the first protection mechanism, it is possible to hide the vast majority of miRNAs while retaining a fair level of prediction accuracy. This is typically very useful in the clinical setting where medical practitioners already know the miRNAs to test for predicting a specific disease. However, such a privacy-preserving mechanism could dramatically jeop-

ardize miRNA utility for biomedical research. Indeed, as we have seen in our previous experiments, the majority of miRNAs need to be masked in order to gain a significant amount of unlinkability, which is not possible if researchers want to test for associations between miRNAs and diseases. Therefore, we additionally present and study a countermeasure where contributors of miRNA expressions directly apply random noise to their vectors of expression levels before providing them to the research community (possibly online), in a fully distributed manner (i.e., independently of other contributors).

The idea behind adding noise to the raw expression data is to provide indistinguishability between different expression vectors and consequently reduce the tracking capabilities of the adversary. Following the generalized notion of differential privacy [15] previously applied to location privacy [10], we state that a mechanism K achieves *epigeno-indistinguishability* if and only if, for all m -miRNA expression vectors $\mathbf{r}_1, \mathbf{r}_2$,

$$Pr(K(\mathbf{r}_1) \in \mathcal{S}) \leq exp(\epsilon d_2(\mathbf{r}_1, \mathbf{r}_2)) \times Pr(K(\mathbf{r}_2) \in \mathcal{S}),$$

where \mathcal{S} is any subset of the set of possible responses and $d_2(\cdot, \cdot)$ denotes the Euclidean distance. In the following, we assume the set of possible responses lies in the same m -dimensional real-valued space \mathbb{R}^m as the set of original expression vectors. Before defining our mechanism $K(\cdot)$ for achieving epigeno-indistinguishability, let us first give some intuition about the mechanism. The noise mechanism is such that the probability of reporting a noisy expression vector $K(\mathbf{r})$ differs by at most a factor $exp(\epsilon d_2(\mathbf{r}_1, \mathbf{r}_2))$ when the actual, non-obfuscated miRNA expression vectors are \mathbf{r}_1 and \mathbf{r}_2 . This can be achieved by relying on the multivariate Laplacian mechanism that adds noise \mathbf{x} according to the following probability density function $g(\mathbf{x}) = \frac{1}{\alpha} e^{-\epsilon \|\mathbf{x}\|_2}$, where α is a normalization factor ensuring that the integral over all $\mathbf{x} \in \mathbb{R}^m$ equals one.

Sampling noise from the distribution $g(\mathbf{x})$ can be carried out efficiently by generalizing the method used for the planar Laplacian mechanism in [10]. First, we sample the magnitude $\|\mathbf{x}\|_2$ of the noise from a gamma distribution with shape m and scale $1/\epsilon$. Second, we randomly generate the direction $\hat{\mathbf{x}} = \mathbf{x}/\|\mathbf{x}\|_2$ of the noise by uniformly sampling points on the surface \mathbb{S}^{m-1} of a hypersphere [36]. To do so, we can generate m independent Gaussian random variables y_1, y_2, \dots, y_m , and let $\hat{y}_i = y_i / \sqrt{y_1^2 + \dots + y_m^2}$ for $i = 1, \dots, m$. Then the distribution of the vector $\hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_m)$ is uniform over the surface \mathbb{S}^{m-1} , and thus we can set the direction $\hat{\mathbf{x}} := \hat{\mathbf{y}}$. Each person i contributing his miRNA expression profile \mathbf{r}_i will then share, instead of the actual expression data, the noisy vector $K(\mathbf{r}_i) = \mathbf{r}_i + \mathbf{x}$, where \mathbf{x} is independently generated for all participants $i = 1, \dots, n$.

Following this approach, in our evaluation, we first add noise to our dataset of 1000+ individuals (considering only the 446 miRNAs as before). Then, in the second step, we calculate the p -values on the noised data (since the researchers would be provided with exactly this data) and train the SVM the same way as in the previous subsections by subsequently adding miRNAs in the order of their p -values. Similarly, we evaluate the success of our attack on the athletes' dataset, when considering the same 446 miRNAs, but after adding noise. Moreover, we repeat both our experiments 50 times and average the results over all runs.

Figure 7 shows the evolution of the SVM accuracy and linkability (success of the attack), with respect to the amount of noise, tuned by ϵ , that is added to each contributor's miRNA expression profile. As privacy is measured on the same dataset for all six figures, its evolves in a very similar way. Even if the noise is randomly generated, the differences average out with the Monte Carlo method we use. We clearly see that with $\epsilon = 1$, there is almost no privacy gain compared to the attack without countermeasure, whereas for $\epsilon = 0.001$, the attack's success drops by almost 90%. Of course, as for the first countermeasure, there is a utility-privacy trade-off to be found between these two extreme values.

In Figure 7(a), we can observe that, for pancreatitis, $\epsilon = 0.075$ is a good trade-off, with an accuracy decrease of only 0.8% and an unlinkability improvement of 40%. For glioma (Figure 7(b)), the best trade-off is certainly at $\epsilon = 0.05$, with an accuracy decrease of 1.2% and an unlinkability improvement of 51%. For multiple sclerosis, we reach the best trade-off at $\epsilon = 0.025$ with an accuracy decrease of 0.65% and an unlinkability improvement of 63%. For tumor of stomach, we can reach an accuracy decrease of only 0.2% and still improve the unlinkability by as much as 70% with $\epsilon = 0.01$. For renal cancer, we have to sacrifice a bit more of utility, 2.3%, for a privacy

increase of 61%, with $\epsilon = 0.025$. The only disease for which it is quite difficult to get both satisfactory unlinkability and excellent accuracy is melanoma (Figure 7(e)). This is consistent with the hiding mechanism presented in Subsection 6.2, where we observed (in Figure 5(e)) a fast and sharp increase in the linkability attack's success.

6.4 Comparison of Protection Mechanisms

In order to compare both approaches, we decide upon a utility or a privacy requirement, fix it, and then evaluate the best privacy, respectively utility, achieved with both countermeasures. We carry out this evaluation on all 19 diseases for different requirements of utility and privacy.

First, we start by fixing the utility, more precisely the relative accuracy decrease compared to the baseline accuracy. The privacy is measured in terms of the decrease in the matching attack's success. For a given maximal decrease in accuracy $\Delta_{\text{acc}}^{\max}$, we select the optimal number of miRNAs m^* and the optimal amount of noise ϵ^* that maximize the privacy increase Δ_{priv}^m and $\Delta_{\text{priv}}^\epsilon$. In case of the hiding mechanism, we select $m^* = \arg \max_m \Delta_{\text{priv}}^m$ such that $\Delta_{\text{acc}}^m < \Delta_{\text{acc}}^{\max}$. In case of the noise mechanism, we select $\epsilon^* = \arg \max_\epsilon \Delta_{\text{priv}}^\epsilon$ such that $\Delta_{\text{acc}}^\epsilon < \Delta_{\text{acc}}^{\max}$, respectively.

Considering $\Delta_{\text{acc}}^{\max} \in \{0.5\%, 1\%, 2\%, 3\%, 4\%, 5\%\}$ for all 19 diseases, we mostly experience that the noise mechanism provides a better privacy improvement compared to hiding a subset of miRNAs (all results are in Table 2 in the appendix). In particular, 90 out of 114 cases (combinations of disease and $\Delta_{\text{acc}}^{\max}$) yield a better privacy with the noise mechanism. When examining a maximal decrease in accuracy of 2%, the hiding technique provides a better privacy for only 2 diseases, namely glioma and renal cancer. Interestingly, these two diseases stand out also for other values of the maximal accuracy decrease, providing better privacy with the hiding technique in 10 out of 12 cases. However, for all other diseases, adding noise in a distributed manner to individual expression profiles provides better utility for similar levels of privacy. For example, for lung cancer, we are able to achieve an increase in privacy of 79.3% while maintaining a decrease in accuracy of 0.8% using noise with $\epsilon = 0.005$. The best we can achieve for the hiding technique here is either a decrease in accuracy of 0.97% and an increase in privacy of only 46.2% or a larger decrease in accuracy of 1.9% and a privacy improvement of only 50%.

Next, we discuss the results for a fixed minimal improvement of the privacy and compare the corresponding minimal decrease in accuracy in both countermeasures. We now fix the minimal increase in privacy (i.e., the minimal decrease in the attack's success) $\Delta_{\text{priv}}^{\min}$ and minimize the decrease in accuracy: $\arg \min_m \Delta_{\text{acc}}^m$ such

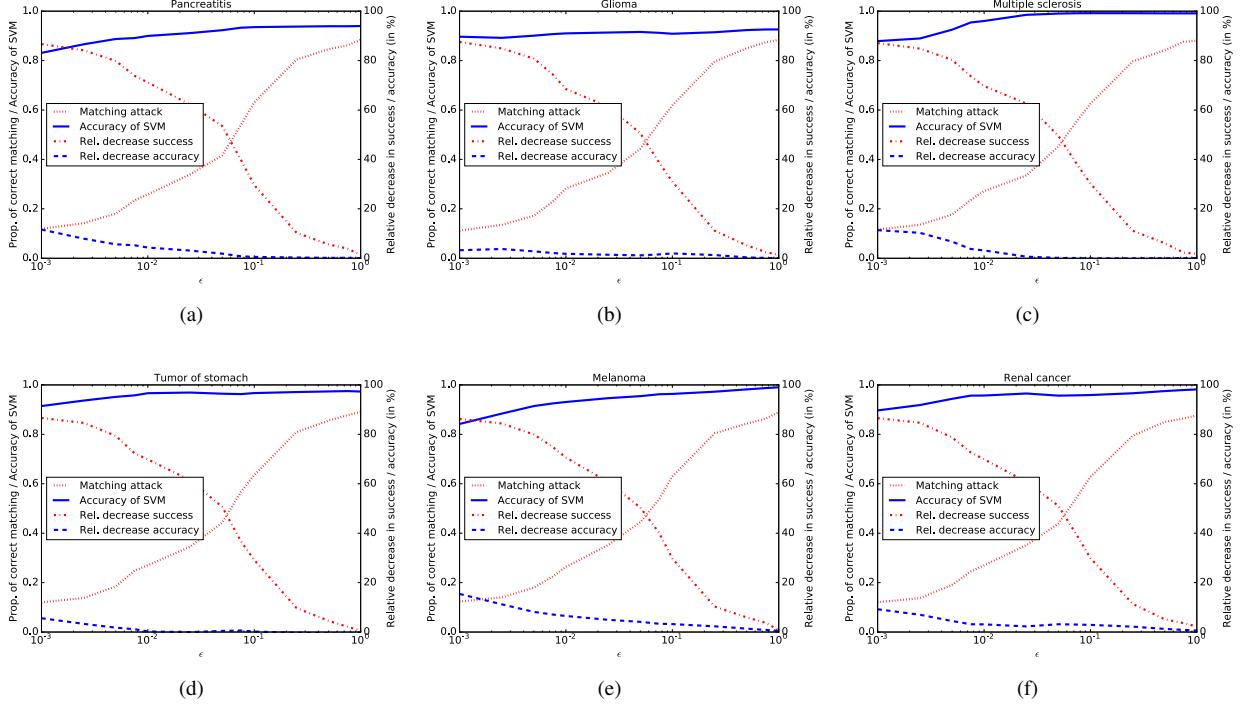


Figure 7: Evolution of privacy and utility (classifier accuracy) plotted against the noise (tuned by ϵ) added to the individual miRNA expression profiles, for the following diseases: (a) Pancreatitis, (b) Glioma, (c) Multiple sclerosis, (d) Tumor of stomach, (e) Melanoma, (f) Renal cancer.

that $\Delta_{\text{priv}}^m > \Delta_{\text{priv}}^{\min}$ and $\arg \min_{\epsilon} \Delta_{\text{acc}}^{\epsilon}$ such that $\Delta_{\text{priv}}^{\epsilon} > \Delta_{\text{priv}}^{\min}$, respectively. We run experiments for values of $\Delta_{\text{priv}}^{\min}$ from 10% up to 90%, in steps of 10% (all results are provided in Table 3 in the appendix).

We again observe that, for most of the evaluated cases, the achieved accuracy is better when adding noise than when hiding miRNAs. In particular, this holds true for 143 out of 171 cases, clear exceptions being again glioma and renal cancer. For those two diseases, the hiding technique provides better accuracy than the noise mechanism in 87.5% of the cases. When fixing the minimal increase in privacy to 70%, only these two diseases provide better results with the hiding technique. For instance, with renal cancer, we achieve 60.8% improvement in privacy with a decrease in accuracy of 2.3% using noise with $\epsilon = 0.025$, whereas we can obtain an increase in privacy of 69.2% and a decrease in accuracy of only 1.7% when using the hiding technique. For the majority of diseases, however, it is clearly the noise mechanism that provides much higher utility. For example, for lung cancer, an increase in privacy of at least 70% is achievable with a decrease in accuracy of only 0.2% with the noise mechanism, while the hiding technique yields a decrease in accuracy of 11.2%.

In summary, we find that the noise mecha-

nism presented in Section 6.3, providing epigeno-indistinguishability, is able to achieve a better privacy-utility trade-off than the hiding mechanism for the vast majority of studied diseases (17 out of 19). We have also shown in Section 6.2 that the privacy improvement with the hiding mechanism could actually be too optimistic due to the correlations existing between miRNAs. This is another argument to favor the noise mechanism rather than the hiding technique. Moreover, the p -values used to rank the miRNAs in the hiding mechanism actually require that, at some point in time, some entity, gets access to the full set of miRNAs of a significant number of individuals in order to measure these p -values. The noise mechanism is fully distributed and does not need to rely on a trusted entity at any point in time. Finally, it allows for more flexibility as it enables, e.g., the biomedical research community to access all miRNA expression levels of contributors.

7 Related Work

We start with the literature highlighting new privacy issues stemming from various types of biomedical data. Schadt et al. have shown that RNA expression data could be used to accurately predict genotypes [48]. The au-

thors present a Bayesian framework that relies on the association existing between expression levels of thousands of genes and genomic variations called expression quantitative trait loci (eQTLs). In the same vein, Philibert et al. demonstrate how methylation array data can be used to construct individually identifying genetic profiles, and to infer substance-use histories, such as alcohol or smoking [45]. Dyke et al. also study privacy risks related to methylation data, and discuss various methods to balance data open-access and (epi)genomic privacy [18]. Franzosa et al. evaluate how different samples of human microbiomes can be linked over time [23]. Their results show that more than 80% of individuals can still be uniquely identified one year later. Fierer et al. had already provided some evidence on the feasibility of linking skin bacterial communities back in 2010, but with very few individuals [22].

There has been quite a lot of work on determining membership of individuals in datasets, which is different from linking them over time among different datasets. Also, these previous works focus on genomic data only. Specifically, the attack aims to identify a victim’s participation in a genome-wide association study (GWAS) based on aggregate statistics on the GWAS dataset, knowing the victim’s genome (or part of it). Homer et al. are the first to thoroughly assess the feasibility and robustness of such an attack by relying upon statistics such as allele frequency or genotype counts [27]. Wang et al. extend the initial attack by making use of the correlations among the different positions in the genome [52]. Their attack proves to be effective with the statistics related to only a few hundreds genetic variants. Im et al. show that, if the victim’s phenotype is rather extreme or if multiple phenotypes are available, regression coefficients can reveal the victim’s participation in a genome-wide association study as much as allele frequencies [31]. Dwork et al. have very recently demonstrated the robustness of such an attack on distorted summary statistics [17].

On the protection side, various papers have studied how to apply noise to summary statistics to protect the privacy of GWAS participants. Johnson and Shmatikov design and implement algorithms for accurate and differentially private computation of various statistics of interest, such as the location of the most significant genomic variants, or the p -values of statistical tests between a given variant and the associated diseases [32]. Uhler et al. have also proposed to rely upon differential privacy for sharing GWAS results privately. In [51], they present methods for privately disclosing allele frequencies, chi-square statistics, and p -values. In [54], Yu et al. extend these methods by allowing for arbitrary number of cases and controls, assess their performance and compare it with the mechanism proposed by Johnson and Shmatikov. In [55], Yu et al. present a differentially-

private mechanism for logistic regression and show how it can be applied to the analysis of GWAS data. In the pharmacogenetic context, Fredrikson et al. show that differential privacy mechanisms can induce bad warfarin dosing, thus expose patients to increased risk of stroke, bleeding events, and mortality [24]. Tramèr et al. [50] investigate how a relaxation of differential privacy that considers more reasonable amounts of background knowledge can help reach a better privacy-utility trade-off for releasing differentially private chi-square statistics in GWAS.

Our work differs from these in the sense that one of our protection mechanisms directly applies noise on the raw miRNA data to guarantee a certain degree of indistinguishability between them, instead of adding noise to summary statistics to ensure differential privacy. Our second defense technique relies on sharing a subset of miRNA data, which is closer to what Humbert et al. have developed in the genomic-privacy context. In particular, they propose an optimization algorithm that enables to share raw genomic variants (rather than summary statistics), e.g., for research, satisfying the genomic privacy requirements of all individuals in a family [29]. More generally, our work aims to protect real-valued miRNA expression vectors, which vary over time much more than DNA data.

8 Conclusion

To the best of our knowledge, this work is the very first to demonstrate that personal miRNA expression profiles can be successfully tracked over time. Our study sheds light on a widely overlooked problem, namely privacy risks stemming from epigenetic data, and brings this issue to the attention of both the biomedical and computer security research communities. In addition to the in-depth evaluation of the temporal linkability of miRNA expression profiles, we propose two defense mechanisms based on well-established privacy-enhancing methods: (i) hiding a subset of the expression data, and (ii) adding noise to the released expression profiles. We thoroughly evaluate the impact of these countermeasures on biomedical utility by studying how much accuracy decrease they induce in a typical machine-learning algorithm for predicting diseases. We observe that, for the majority of the 19 diseases studied in our experiments, the noise mechanism provides a better privacy-utility trade-off than the hiding method. Moreover, we highlight that the noise mechanism can be applied directly by the data contributors, independently of other contributors, and provides more flexibility for the biomedical community. Our work demonstrates that achieving indistinguishability by adding noise is a promising technique that could be applied to other types of biomedical data in the future.

Our results provide enough evidence about the extent of the threat to remove miRNA expression data from publicly accessible databases. Due to the limited number of individuals present in our datasets, we could not rely on supervised learning algorithms, which would certainly further improve the tracking capabilities of the adversary. We hope that this work will lead to further research on better understanding and protecting the privacy of miRNA expression data. Considering larger databases or uncertain membership of participants in the targeted databases are other promising directions for follow-up work.

9 Acknowledgements

This work has been partially funded by the German Research Foundation (DFG) via the collaborative research center “Methods and Tools for Understanding and Controlling Privacy” (SFB 1223), project A5.

References

- [1] Arrayexpress. <https://www.ebi.ac.uk/arrayexpress>. Accessed: 2016-02-12.
- [2] Bilans de santé en balade sur le net. <http://www.lematin.ch/suisse/bilans-sante-balade-net/story/21621328>. Accessed: 2016-02-03.
- [3] The black market for stolen health care data. <http://www.npr.org/sections/alltechconsidered/2015/02/13/385901377/the-black-market-for-stolen-health-care-data>. Accessed: 2016-02-03.
- [4] Gene expression omnibus. <http://www.ncbi.nlm.nih.gov/geo>. Accessed: 2016-02-12.
- [5] Health insurer anthem discloses customer and employee data breach. <http://www.computerworld.com/article/2879649/health-insurer-anthem-discloses-customer-and-employee-data-breach.html>. Accessed: 2016-02-03.
- [6] Medical data - a new target for hackers. <https://www.logpoint.com/se/about-us/blog/249-medical-data-a-new-target-for-hackers>. Accessed: 2016-02-03.
- [7] micrornas: Definition and overview. <https://www.thermofisher.com/de/de/home/references/ambion-tech-support/microrna-studies/tech-notes/micrornas-definition-and-overview.html>. Accessed: 2016-02-12.
- [8] Premera, anthem data breaches linked by similar hacking tactics. <http://www.computerworld.com/article/2898419/data-breach/premera-anthem-data-breaches-linked-by-similar-hacking-tactics.html>. Accessed: 2016-02-03.
- [9] Urgent probe as michael schumacher's medical records stolen and put on sale for 40k. <http://www.express.co.uk/news/world/484495/Investigation-underway-after-Michael-Schumacher-s-medical-records-stolen>. Accessed: 2016-02-03.
- [10] M. E. Andrés, N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 901–914. ACM, 2013.
- [11] E. Ayday, E. De Cristofaro, J.-P. Hubaux, and G. Tsudik. Whole genome sequencing: Revolutionary medicine or privacy nightmare? *Computer*, pages 58–66, 2015.
- [12] C. Backes, P. Leidinger, A. Keller, M. Hart, T. Meyer, E. Meese, and A. Hecksteden. Blood born mirnas signatures that can serve as disease specific biomarkers are not significantly affected by overall fitness and exercise. *PloS one*, 9(7):e102183, 2014.
- [13] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.
- [14] C. Cachin. *Entropy measures and unconditional security in cryptography*. PhD thesis, SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH, 1997.
- [15] K. Chatzikokolakis, M. E. Andrés, N. E. Bordenabe, and C. Palamidessi. Broadening the scope of differential privacy using metrics. In *Privacy Enhancing Technologies*, pages 82–102. Springer, 2013.
- [16] J. Cloud. Why your DNA isn't your destiny. *Time*, January 2010.
- [17] C. Dwork, A. Smith, T. Steinke, J. Ullman, and S. Vadhan. Robust traceability from trace amounts. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 650–669. IEEE, 2015.
- [18] S. O. Dyke, W. A. Cheung, Y. Joly, O. Ammerpohl, P. Lutsik, M. A. Rothstein, M. Caron, S. Busche, G. Bourque, L. Rönnblom, et al. Epigenome data release: a participant-centered approach to privacy protection. *Genome biology*, 16:1–12, 2015.
- [19] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.
- [20] Y. Erlich and A. Narayanan. Routes for breaching and protecting genetic privacy. *Nature Reviews Genetics*, 15:409–421, 2014.
- [21] A. P. Feinberg and M. D. Fallin. Epigenetics at the crossroads of genes and the environment. *JAMA*, 314:1129–1130, 2015.
- [22] N. Fierer, C. L. Lauber, N. Zhou, D. McDonald, E. K. Costello, and R. Knight. Forensic identification using skin bacterial communities. *Proceedings of the National Academy of Sciences*, 107(14):6477–6481, 2010.
- [23] E. A. Franzosa, K. Huang, J. F. Meadow, D. Gevers, K. P. Lemon, B. J. Bohannan, and C. Huttenhower. Identifying personal microbiomes using metagenomic codes. *Proceedings of the National Academy of Sciences*, page 201423854, 2015.
- [24] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 17–32, 2014.
- [25] R. C. Friedman, K. K.-H. Farh, C. B. Burge, and D. P. Bartel. Most mammalian mRNAs are conserved targets of microRNAs. *Genome research*, 19(1):92–105, 2009.
- [26] M. Gymrek, A. L. McGuire, D. Golan, E. Halperin, and Y. Erlich. Identifying personal genomes by surname inference. *Science*, 339:321–324, 2013.
- [27] N. Homer, S. Szelinger, M. Redman, D. Duggan, W. Tembe, J. Muehling, J. V. Pearson, D. A. Stephan, S. F. Nelson, and D. W. Craig. Resolving individuals contributing trace amounts of dna to highly complex mixtures using high-density snp genotyping microarrays. *PLoS Genet*, 4(8):e1000167, 2008.

- [28] M. Humbert, E. Ayday, J.-P. Hubaux, and A. Telenti. Addressing the concerns of the Lacks family: quantification of kin genomic privacy. In *Proceedings of the 2013 ACM SIGSAC CCS*, pages 1141–1152, 2013.
- [29] M. Humbert, E. Ayday, J.-P. Hubaux, and A. Telenti. Reconciling utility with privacy in genomics. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 11–20. ACM, 2014.
- [30] M. Humbert, K. Huguenin, J. Hugonot, E. Ayday, and J.-P. Hubaux. De-anonymizing genomic databases using phenotypic traits. *Proceedings on Privacy Enhancing Technologies(PoPETs)*, 2015.
- [31] H. K. Im, E. R. Gamazon, D. L. Nicolae, and N. J. Cox. On sharing quantitative trait gwas results in an era of multiple-omics data and the limits of genomic privacy. *The American Journal of Human Genetics*, 90(4):591–598, 2012.
- [32] A. Johnson and V. Shmatikov. Privacy-preserving data exploration in genome-wide association studies. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1079–1087. ACM, 2013.
- [33] P. A. Jones and S. B. Baylin. The epigenomics of cancer. *Cell*, 128:683–692, 2007.
- [34] A. Keller, P. Leidinger, A. Bauer, A. ElSharawy, J. Haas, C. Backes, A. Wendschlag, N. Giese, C. Tjaden, K. Ott, et al. Toward the blood-borne mirnyme of human diseases. *Nature methods*, 8:841–843, 2011.
- [35] A. Keller, P. Leidinger, B. Vogel, C. Backes, A. ElSharawy, V. Galata, S. C. Mueller, S. Marquart, M. G. Schrauder, R. Strick, et al. mirnas can be generally associated with human pathologies as exemplified for mir-144*. *BMC medicine*, 12(1):224, 2014.
- [36] F. Koufogiannis, S. Han, and G. J. Pappas. Optimality of the laplace mechanism in differential privacy. *arXiv preprint arXiv:1504.00065*, 2015.
- [37] P. Leidinger, C. Backes, S. Deutscher, K. Schmitt, S. C. Mueller, K. Frese, J. Haas, K. Ruprecht, F. Paul, C. Stähler, et al. A blood based 12-mirna signature of alzheimer disease patients. *Genome Biol*, 14:R78, 2013.
- [38] P. Leidinger, V. Galata, C. Backes, C. Stähler, S. Rheinheimer, H. Huwer, E. Meese, and A. Keller. Longitudinal study on circulating mirnas in patients after lung cancer resection. *Oncotarget*, 6:16674, 2015.
- [39] Z. Lin, A. B. Owen, and R. B. Altman. Genomic research and human subject privacy. *SCIENCE-NEW YORK THEN WASHINGTON-*, pages 183–183, 2004.
- [40] J. Lu, G. Getz, E. A. Miska, E. Alvarez-Saavedra, J. Lamb, D. Peck, A. Sweet-Cordero, B. L. Ebert, R. H. Mak, A. A. Ferrando, et al. Microrna expression profiles classify human cancers. *nature*, 435(7043):834–838, 2005.
- [41] J. L. Massey. Guessing and entropy. In *Information Theory, 1994. Proceedings., 1994 IEEE International Symposium on*, page 204. IEEE, 1994.
- [42] M. Naveed, E. Ayday, E. W. Clayton, J. Fellay, C. A. Gunter, J.-P. Hubaux, B. A. Malin, and X. Wang. Privacy in the genomic era. *ACM Computing Surveys (CSUR)*, 48:6, 2015.
- [43] T. Ngun et al. Abstract: A novel predictive model of sexual orientation using epigenetic markers. In *American Society of Human Genetics 2015 Annual Meeting*, 2015.
- [44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [45] R. A. Philibert, N. Terry, C. Erwin, W. J. Philibert, S. R. Beach, and G. H. Brody. Methylation array data can simultaneously identify individuals and convey protected health information: an unrecognized ethical concern. *Clinical epigenetics*, 6:28, 2014.
- [46] I. A. Qureshi and M. F. Mehler. Advances in epigenetics and epigenomics for neurodegenerative diseases. *Current neurology and neuroscience reports*, 11:464–473, 2011.
- [47] M. A. Rothstein, Y. Cai, and G. E. Marchant. The ghost in our genes: legal and ethical implications of epigenetics. *Health matrix (Cleveland, Ohio: 1991)*, 19:1, 2009.
- [48] E. E. Schadt, S. Woo, and K. Hao. Bayesian method to predict individual snp genotypes from gene expression data. *Nature genetics*, 44:603–608, 2012.
- [49] M. E. Tipping and C. M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999.
- [50] F. Tramèr, Z. Huang, J.-P. Hubaux, and E. Ayday. Differential privacy with bounded priors: reconciling utility and privacy in genome-wide association studies. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1286–1297. ACM, 2015.
- [51] C. Uhler, A. Slavković, and S. E. Fienberg. Privacy-preserving data sharing for genome-wide association studies. *The Journal of privacy and confidentiality*, 5(1):137, 2013.
- [52] R. Wang, Y. F. Li, X. Wang, H. Tang, and X. Zhou. Learning your identity and disease from research papers: information leaks in genome wide association study. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 534–544. ACM, 2009.
- [53] L. D. Wood, D. W. Parsons, S. Jones, J. Lin, T. Sjöblom, R. J. Leary, D. Shen, S. M. Boca, T. Barber, J. Ptak, et al. The genomic landscapes of human breast and colorectal cancers. *Science*, 318:1108–1113, 2007.
- [54] F. Yu, S. E. Fienberg, A. B. Slavković, and C. Uhler. Scalable privacy-preserving data sharing methodology for genome-wide association studies. *Journal of biomedical informatics*, 50:133–141, 2014.
- [55] F. Yu, M. Rybar, C. Uhler, and S. E. Fienberg. Differentially-private logistic regression for detecting multiple-snp association in gwas databases. In *Privacy in Statistical Databases*, pages 170–184. Springer, 2014.

A Appendix

A.1 Human Subjects and Ethical Considerations

The studies have received an approval from our institutional ethics review board. Moreover, not only have all datasets been stored and analyzed in anonymized form, but we also handled our results with great care to not deanonymize any of the patients. This way, we ensured that all participants were treated equally and with respect.

$\Delta_{\text{acc}}^{\max}$	0.5%		1.0%		2.0%		3.0%		4.0%		5.0%	
Disease	Δ_{priv}^m	$\Delta_{\text{priv}}^\epsilon$										
Periodontitis	26.9%	74.1%	26.9%	79.2%	50.0%	79.2%	88.5%	83.6%	88.5%	83.6%	88.5%	83.6%
Renal cancer	30.8%	-	30.8%	3.6%	69.2%	5.2%	73.1%	60.8%	73.1%	72.7%	80.8%	78.8%
Wilms tumor	3.8%	6.4%	7.7%	9.5%	7.7%	40.1%	7.7%	61.5%	7.7%	70.4%	11.5%	74.3%
Benign prostate hyperplasia	-3.8%	10.6%	3.8%	70.5%	11.5%	72.2%	46.2%	79.2%	57.7%	79.2%	65.4%	79.2%
Chronic obstructive pulmonary disease (COPD)	0.0%	2.7%	0.0%	5.5%	0.0%	12.5%	0.0%	12.5%	15.4%	50.3%	23.1%	69.8%
Colon cancer	19.2%	11.4%	30.8%	30.5%	30.8%	60.2%	57.7%	60.2%	73.1%	70.5%	73.1%	73.8%
Ductal adenocarcinoma	0.0%	50.6%	3.8%	50.6%	7.7%	62.5%	42.3%	62.5%	50.0%	69.5%	50.0%	74.2%
Glioma	65.4%	5.2%	65.4%	5.2%	80.8%	68.5%	80.8%	80.8%	80.8%	87.5%	80.8%	87.5%
Lung cancer	11.5%	74.1%	46.2%	79.3%	50.0%	79.3%	50.0%	79.3%	50.0%	79.3%	50.0%	79.3%
Melanoma	0.0%	-	0.0%	3.8%	0.0%	5.9%	3.8%	10.3%	38.5%	40.2%	38.5%	60.7%
Multiple sclerosis	19.2%	49.5%	53.8%	62.6%	53.8%	62.6%	61.5%	62.6%	61.5%	73.7%	61.5%	73.7%
Myocardial infarction	3.8%	52.4%	3.8%	52.4%	3.8%	60.5%	38.5%	60.5%	42.3%	74.6%	42.3%	74.6%
Non-ischaemic systolic heart failure	-3.8%	80.0%	0.0%	80.0%	46.2%	80.0%	46.2%	84.7%	46.2%	84.7%	46.2%	84.7%
Ovarian cancer	26.9%	78.5%	26.9%	78.5%	42.3%	78.5%	42.3%	84.3%	50.0%	84.3%	50.0%	86.2%
Pancreatitis	19.2%	10.3%	26.9%	39.8%	26.9%	53.5%	26.9%	53.5%	57.7%	62.2%	65.4%	71.2%
Prostate cancer	-3.8%	-	-3.8%	-	3.8%	4.0%	42.3%	6.2%	42.3%	10.1%	42.3%	38.5%
Psoriasis	0.0%	6.5%	0.0%	31.4%	3.8%	74.0%	19.2%	80.1%	23.1%	80.1%	61.5%	80.1%
Sarcoidosis	0.0%	69.3%	3.8%	74.0%	50.0%	79.8%	92.3%	79.8%	92.3%	79.8%	92.3%	79.8%
Tumor of stomach	15.4%	69.8%	34.6%	69.8%	65.4%	79.4%	65.4%	79.4%	65.4%	84.6%	65.4%	84.6%

Table 2: Relative increase in privacy for both defense mechanisms in relation to a fixed maximal decrease in accuracy. “-” means that the respective maximal decrease in accuracy was not achievable with any ϵ we tested for. A negative value means that the attack’s success rate could, in this case, even exceed the success rate with all miRNAs taken into account.

$\Delta_{\text{priv}}^{\min}$	30.0%		40.0%		50.0%		60.0%		70.0%		80.0%	
Disease	Δ_{acc}^m	$\Delta_{\text{acc}}^\epsilon$										
Periodontitis	1.9%	-1.9%	1.9%	-1.9%	2.6%	-1.9%	2.6%	-1.9%	2.6%	-0.8%	2.6%	2.9%
Renal cancer	0.0%	2.3%	1.7%	2.3%	1.7%	2.3%	1.7%	2.3%	2.5%	3.1%	4.8%	7.0%
Wilms tumor	5.2%	1.4%	5.2%	1.7%	5.5%	2.2%	5.5%	2.8%	8.1%	3.2%	15.5%	11.3%
Benign prostate hyperplasia	2.7%	0.5%	2.7%	0.6%	3.5%	0.6%	5.0%	0.9%	5.6%	0.9%	5.6%	5.5%
Chronic obstructive pulmonary disease (COPD)	7.9%	3.3%	12.0%	3.3%	12.0%	3.3%	15.4%	4.1%	15.6%	5.3%	15.6%	9.0%
Colon cancer	0.7%	0.8%	2.4%	1.3%	2.4%	1.3%	3.3%	1.9%	3.9%	3.3%	7.7%	9.4%
Ductal adenocarcinoma	2.8%	0.1%	2.8%	0.4%	5.2%	0.4%	5.2%	1.8%	6.4%	4.6%	6.4%	6.4%
Glioma	0.0%	1.2%	0.0%	1.2%	0.4%	1.2%	0.4%	1.4%	1.1%	2.1%	1.1%	2.8%
Lung cancer	0.7%	-1.5%	1.0%	-1.5%	6.6%	-1.5%	8.1%	-1.5%	11.2%	0.2%	18.2%	5.5%
Melanoma	3.7%	3.4%	5.0%	3.4%	7.5%	4.1%	7.5%	4.9%	7.5%	6.5%	10.0%	11.2%
Multiple sclerosis	0.9%	-0.0%	0.9%	0.1%	0.9%	0.7%	2.3%	0.7%	8.1%	3.8%	8.1%	6.7%
Myocardial infarction	2.8%	0.0%	3.6%	0.4%	7.2%	0.4%	7.3%	1.3%	7.3%	3.3%	11.2%	6.7%
Non-ischaemic systolic heart failure	2.0%	-2.6%	2.0%	-2.6%	8.5%	-2.1%	8.5%	-2.1%	9.3%	-1.5%	9.3%	2.5%
Ovarian cancer	1.3%	-0.7%	1.3%	-0.7%	5.5%	-0.7%	6.7%	-0.7%	9.0%	-0.7%	9.0%	2.5%
Pancreatitis	3.8%	0.8%	3.8%	1.9%	3.8%	1.9%	4.5%	3.1%	7.9%	4.3%	7.9%	7.9%
Prostate cancer	2.7%	4.8%	2.7%	5.0%	7.6%	5.0%	7.6%	5.6%	7.6%	5.6%	11.5%	8.9%
Psoriasis	4.3%	1.0%	4.3%	1.3%	4.3%	1.3%	4.3%	1.4%	5.8%	1.4%	10.0%	2.1%
Sarcoidosis	1.4%	-0.2%	1.6%	-0.2%	2.2%	-0.2%	2.2%	-0.2%	2.2%	0.6%	2.2%	5.3%
Tumor of stomach	0.9%	-0.0%	1.7%	-0.0%	2.0%	-0.0%	2.0%	-0.0%	5.1%	1.1%	5.1%	3.3%

Table 3: Relative decrease in accuracy for both defense mechanisms in relation to a fixed minimal increase in privacy. A negative value means that the accuracy could, in this case, even exceed the baseline accuracy (utility).